Table of Contents

数据结构和算法 Algorithm and Data Structure Notes

Problems coming from LeetCode, LintCode, TopCoder, CtCi, etc.

持续更新中...
Under construction...

Contents

Linked List

Linked List 基本操作

Based on Java

ListNode Class

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int x) { val = x; }
}
```

Reverse a Linked List

```
public ListNode reverse(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode next = head.next;
        head.next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

Find the Middle Point

```
public ListNode findMiddle(ListNode head) {
    ListNode fast = head;
    ListNode slow = head.next;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}
```

Find the Nth Element

```
ListNode fast = head;
for (int i = 0; i < n - 1; i++) {
    fast = fast.next;
    if (fast == null) {
        return null;
    }
}
```

Dummy Node

```
ListNode dummy = new ListNode(Integer.MIN_VALUE);
ListNode current = dummy;
...
while (current != null) {
    ...
    current.next = blahblahblah;
    ...
    current = current.next;
}
...
return dummy.next;
```

Merge Two Sorted Lists

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode lastNode = dummy;

    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            lastNode.next = l1;
            l1 = l1.next;
        } else {
            lastNode.next = l2;
            l2 = l2.next;
        }
        lastNode = lastNode.next;
    }

    if (l1 != null) {
        lastNode.next = l1;
    } else {
        lastNode.next = l2;
    }

    return dummy.next;
}
```

Linked List Has Cycle

```
public Boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) {
        return false;
    }

    ListNode fast, slow;
    fast = head.next;
    slow = head;
    while (fast != slow) {
        if(fast==null || fast.next==null)
            return false;
        fast = fast.next.next;
        slow = slow.next;
    }
    return true;
}
```

Sort List

Question

leetcode: Sort List | LeetCode OJ lintcode: (98) Sort List

```
Sort a linked list in O(n log n) time using constant space compl
exity.
```

题解思路

链表的排序操作，对于常用的排序算法，能达到O(nlogn)的复杂度有快速排序(平均情况)，归并排序，堆排序。

对于数组，归并排序一般需要使用O(n)的额外空间，虽然也有原地的实现方法。但对于链表这样的数据结构，排序是指针next值的变化，所以只需要常数级的额外空间。

归并排序的核心思想在于，根据分治思想，可以按照左、右、合并的顺序，实现递归模型，也就是先找出左右链表，最后进行归并。

三个主要步骤

1. 找到链表中点：可以通过快慢指针的方法
2. 合并两个有序链表：链表的基本问题
3. 分治递归

源代码

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
```

```
 *         this.next = null;
 *     }
 * }
 */

public class Solution {
    private ListNode findMiddle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }

    private ListNode merge(ListNode head1, ListNode head2) {
        ListNode dummy = new ListNode(0);
        ListNode pointer = dummy;

        while (head1 != null && head2 != null) {
            if (head1.val < head2.val) {
                pointer.next = head1;
                head1 = head1.next;
            } else {
                pointer.next = head2;
                head2 = head2.next;
            }
            pointer = pointer.next;
        }

        if (head1 != null) {
            pointer.next = head1;
        } else {
            pointer.next = head2;
        }

        return dummy.next;
    }
```

```
    /**
     * @param head: The head of linked list.
     * @return: You should return the head of the sorted linked
list,
                    using constant space complexity.
     */
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode mid = findMiddle(head);
        ListNode right = sortList(mid.next);
        mid.next = null;
        ListNode left = sortList(head);

        return merge(left, right);
    }
}
```

Merge k Sorted Lists

Question

leetcode: Merge k Sorted Lists | LeetCode OJ lintcode: Merge k Sorted Lists

```
Merge k sorted linked lists and return it as one sorted list.
Analyze and describe its complexity.
```

题解思路

归并k个已排序链表，可以分解问题拆解成为，重复k次，归并2个已排序链表。不过这种方法会在LeetCode中TLE（超出时间限制）。总共需要k次merge two sorted lists的合并过程。

改进方法：

1. 使用merge sort中的二分的思维，将包含k个链表的列表逐次分成两个部分，再逐次对两个链表合并，这样就有 log(k)次合并过程，每次均使用merge two sorted lists的算法。时间复杂度O(nlog(k))。
2. 使用Priority Queue。这样保持每次取出的节点中是当前最小的，依次加入新的链表，从而得到合并的结果。

源代码

- Divide and Conquer

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
```

```
 *     }
 * }
 */
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode lastNode = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                lastNode.next = l1;
                l1 = l1.next;
            } else {
                lastNode.next = l2;
                l2 = l2.next;
            }
            lastNode = lastNode.next;
        }

        if (l1 != null) {
            lastNode.next = l1;
        } else {
            lastNode.next = l2;
        }

        return dummy.next;
    }
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    public ListNode mergeKListsNaive(List<ListNode> lists) {
        if (lists == null || lists.size() == 0) {
            return null;
        }
        if (lists.size() == 1) {
            return lists.get(0);
        }

        int listSize = lists.size();
```

```
        ListNode base = lists.get(0);

        for (int i = 1; i < listSize; i++) {
            base = mergeTwoLists(base, lists.get(i));
        }
        return base;
    }
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
     public ListNode mergeKLists(List<ListNode> lists) {
         if (lists.size() == 0) {
             return null;
         }
         if (lists.size() == 1) {
             return lists.get(0);
         }
         if (lists.size() == 2) {
             return mergeTwoLists(lists.get(0), lists.get(1));
         }
         return mergeTwoLists(
            mergeKLists(lists.subList(0, lists.size()/2)),
            mergeKLists(lists.subList(lists.size()/2, lists.size
()))
         );
     }
}
```

- Divide & Conquer Another implementation

```
public class Solution {
    /**
     * @param lists: a list of ListNode
     * @return: The head of one sorted list.
     */
    public ListNode mergeKLists(List<ListNode> lists) {
        if (lists.size() == 0) {
```

```
            return null;
        }
        return mergeHelper(lists, 0, lists.size() - 1);
    }

    private ListNode mergeHelper(List<ListNode> lists, int start
, int end) {
        if (start == end) {
            return lists.get(start);
        }

        int mid = start + (end - start) / 2;
        ListNode left = mergeHelper(lists, start, mid);
        ListNode right = mergeHelper(lists, mid + 1, end);
        return mergeTwoLists(left, right);
    }

    private ListNode mergeTwoLists(ListNode list1, ListNode list
2) {
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
        while (list1 != null && list2 != null) {
            if (list1.val < list2.val) {
                tail.next = list1;
                tail = list1;
                list1 = list1.next;
            } else {
                tail.next = list2;
                tail = list2;
                list2 = list2.next;
            }
        }
        if (list1 != null) {
            tail.next = list1;
        } else {
            tail.next = list2;
        }

        return dummy.next;
    }
```

```
}
```

- Heap

```
public class Solution {
    private Comparator<ListNode> ListNodeComparator = new Compar
ator<ListNode>() {
        public int compare(ListNode left, ListNode right) {
            if (left == null) {
                return 1;
            } else if (right == null) {
                return -1;
            }
            return left.val - right.val;
        }
    };

    public ListNode mergeKLists(ArrayList<ListNode> lists) {
        if (lists == null || lists.size() == 0) {
            return null;
        }

        Queue<ListNode> heap = new PriorityQueue<ListNode>(lists
.size(), ListNodeComparator);
        for (int i = 0; i < lists.size(); i++) {
            if (lists.get(i) != null) {
                heap.add(lists.get(i));
            }
        }

        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
        while (!heap.isEmpty()) {
            ListNode head = heap.poll();
            tail.next = head;
            tail = head;
            if (head.next != null) {
                heap.add(head.next);
            }
```

```
            }
            return dummy.next;
        }
    }
```

Linked List Cycle

Question

leetcode: Linked List Cycle lintcode: Linked List Cycle

```
Given a linked list, determine if it has a cycle in it.

Follow up:
Can you solve it without using extra space?
```

题解思路

链表中的Two Pointers是一种很常用的思想，快慢两个指针，通过是否相遇，来判断链表中是否有环。相比于用HashMap的方法，优点是空间复杂度仅为O(1)，时间复杂度O(n)。

参考 LeetCode: https://leetcode.com/articles/linked-list-cycle/

源代码

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */

public class Solution {
```

```
    private ListNode findMiddle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }

    private ListNode merge(ListNode head1, ListNode head2) {
        ListNode dummy = new ListNode(0);
        ListNode pointer = dummy;

        while (head1 != null && head2 != null) {
            if (head1.val < head2.val) {
                pointer.next = head1;
                head1 = head1.next;
            } else {
                pointer.next = head2;
                head2 = head2.next;
            }
            pointer = pointer.next;
        }

        if (head1 != null) {
            pointer.next = head1;
        } else {
            pointer.next = head2;
        }

        return dummy.next;
    }

    /**
     * @param head: The head of linked list.
     * @return: You should return the head of the sorted linked
list,
                    using constant space complexity.
     */
```

```
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode mid = findMiddle(head);
        ListNode right = sortList(mid.next);
        mid.next = null;
        ListNode left = sortList(head);

        return merge(left, right);
    }
}
```

Linked List Cycle II

Question

leetcode: Linked List Cycle II lintcode: Linked List Cycle II

```
Given a linked list, return the node where the cycle begins. If
there is no cycle, return null.

Note: Do not modify the linked list.
```

题解思路

可以先判断Linked List是否有Cycle，利用two pointers可以很快得到。接下来就要分析，快慢指针第一次相遇时，位置、步数和环的关系。

```
    a            b
A ------ B --------+
         |         |
       c |         |
         +-------- C



* A: 起始点
* B: Cycle Begins
* C: 1st 快慢指针相遇点


* A->B: a
* B->C: b
* C->B: c
* 环的长度 (b+c) 为 R
```

第一次相遇时，慢指针所走步数为

a + b

快指针走的步数为

a + b + nR

我们知道快指针是慢指针速度的2倍，因此

2(a + b) = a + b + nR

那么

a + b = nR

同时

b + c = R

所以

a = (n - 1)R + c;

也就是说，从A点和C点同时出发，以相同的速度前进，那么下一次相遇的位置将是B。

参考 LeetCode: https://leetcode.com/articles/linked-list-cycle/

源代码

```
/**
 * Definition for ListNode.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int val) {
 *         this.val = val;
 *         this.next = null;
 *     }
 * }
 */

public class Solution {
```

```
    private ListNode findMiddle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head.next;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }

    private ListNode merge(ListNode head1, ListNode head2) {
        ListNode dummy = new ListNode(0);
        ListNode pointer = dummy;

        while (head1 != null && head2 != null) {
            if (head1.val < head2.val) {
                pointer.next = head1;
                head1 = head1.next;
            } else {
                pointer.next = head2;
                head2 = head2.next;
            }
            pointer = pointer.next;
        }

        if (head1 != null) {
            pointer.next = head1;
        } else {
            pointer.next = head2;
        }

        return dummy.next;
    }

    /**
     * @param head: The head of linked list.
     * @return: You should return the head of the sorted linked
list,
                    using constant space complexity.
     */
```

```
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode mid = findMiddle(head);
        ListNode right = sortList(mid.next);
        mid.next = null;
        ListNode left = sortList(head);

        return merge(left, right);
    }
}
```

Add Two Numbers II

Question

> You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in forward order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.
>
> Example:
>
> Given 6->1->7 + 2->9->5. That is, 617 + 295.
>
> Return 9->1->2. That is, 912.

Analysis

相比于数字为倒序的链表相加问题(add two numbers)，顺序的数字（单向）链表存在处理进位的问题，因为没有反向指针，所以后节点所得的进位，难以加到前一个节点上。 此外，与 add two numbers 问题一样，不能简单地将链表转为整形int或长整型long，因为很有可能超出范围，使用链表的好处正是在于可以几乎无限制地增加number的位数，如果有转换的过程，则很有可能丢失精度。

因此，最简单的解决办法，就是将 add two numbers ii 问题，转化为 add two numbers 问题：先分别对每个number链表进行反转，然后进行相加，最后将所得链表再进行反转。

复杂问题可以通过分析，转化和分解为更小的，更基础的问题。

Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
```

```
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {

    public ListNode reverse(ListNode head) {
        ListNode prev = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }

    public ListNode addLists(ListNode l1, ListNode l2) {
        if (l1 == null && l2 == null) {
            return null;
        }

        ListNode head = new ListNode(0);
        ListNode pointer = head;

        int carry = 0;

        while (l1 != null && l2 != null) {

            int sum = l1.val + l2.val + carry;
            carry = sum / 10;

            pointer.next = new ListNode(sum % 10);
            pointer = pointer.next;
            l1 = l1.next;
            l2 = l2.next;
        }
```

```
        while (l1 != null) {
            int sum = l1.val + carry;
            carry = sum / 10;
            pointer.next = new ListNode(sum % 10);
            pointer = pointer.next;
            l1 = l1.next;
        }

        while (l2 != null) {
            int sum = l2.val + carry;
            carry = sum / 10;
            pointer.next = new ListNode(sum = sum % 10);
            pointer = pointer.next;
            l2 = l2.next;
        }

        if (carry != 0) {
            pointer.next = new ListNode(carry);
        }

        return head.next;
    }
    /**
     * @param l1: the first list
     * @param l2: the second list
     * @return: the sum list of l1 and l2
     */
    public ListNode addLists2(ListNode l1, ListNode l2) {
        l1 = reverse(l1);
        l2 = reverse(l2);

        return reverse(addLists(l1, l2));
    }
}
```

Array

Array Basic Operations

Iterate Through Loop in Java

Five ways to Iterate Through Loop in Java

```
package crunchify.com.tutorial;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * @author Crunchify.com
 */

public class CrunchifyIterateThroughList {

    public static void main(String[] argv) {

        // create list
        List<String> crunchifyList = new ArrayList<String>();

        // add 4 different values to list
        crunchifyList.add("eBay");
        crunchifyList.add("Paypal");
        crunchifyList.add("Google");
        crunchifyList.add("Yahoo");

        // iterate via "for loop"
        System.out.println("==> For Loop Example.");
        for (int i = 0; i < crunchifyList.size(); i++) {
            System.out.println(crunchifyList.get(i));
        }
```

```
        // iterate via "New way to loop"
        System.out.println("\n==> Advance For Loop Example..");
        for (String temp : crunchifyList) {
            System.out.println(temp);
        }

        // iterate via "iterator loop"
        System.out.println("\n==> Iterator Example...");
        Iterator<String> crunchifyIterator = crunchifyList.itera
tor();
        while (crunchifyIterator.hasNext()) {
            System.out.println(crunchifyIterator.next());
        }

        // iterate via "while loop"
        System.out.println("\n==> While Loop Example....");
        int i = 0;
        while (i < crunchifyList.size()) {
            System.out.println(crunchifyList.get(i));
            i++;
        }

        // collection stream() util: Returns a sequential Stream
 with this collection as its source
        System.out.println("\n==> collection stream() util....")
;
        crunchifyList.forEach((temp) -> {
            System.out.println(temp);
        });
    }
}
```

Partition Array

Problem

> Given an array nums of integers and an int k, partition the array (i.e move the elements in "nums") such that:
>
> - All elements < k are moved to the left
> - All elements >= k are moved to the right
> - Return the partitioning index, i.e the first index i nums[i] >= k.
>
> *Notice*
>
> You should do really partition in array nums instead of just counting the numbers of integers smaller than k.
>
> If all elements in nums are smaller than k, then return nums.length

Analysis

根据给定的k，也就是类似于Quick Sort中的pivot，将array从两头进行缩进，时间复杂度 O(n)

Solution

```java
public class Solution {
    private void swap(int i, int j, int[] arr) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
    /**
     *@param nums: The integer array you should partition
     *@param k: As description
     *return: The index after partition
     */
    public int partitionArray(int[] nums, int k) {

        int pl = 0;
        int pr = nums.length - 1;
        while (pl <= pr) {
            while (pl <= pr && nums[pl] < k) {
                pl++;
            }
            while (pl <= pr && nums[pr] >= k) {
                pr--;
            }
            if (pl <= pr) {
                swap(pl, pr, nums);
            }
        }
        return pl;
    }
}
```

Median of two Sorted Arrays

http://www.lintcode.com/en/problem/median-of-two-sorted-arrays/

Question

> There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays.
>
> Example
>
> Given A=[1,2,3,4,5,6] and B=[2,3,4,5], the median is 3.5.
>
> Given A=[1,2,3] and B=[4,5], the median is 3.

Analysis

参考九章中的解释：

对于一个长度为n的已排序数列a，若n为奇数，中位数为a[n / 2 + 1], 若n为偶数，则中位数(a[n / 2] + a[n / 2 + 1]) / 2; 如果我们可以在两个数列中求出第K小的元素，便可以解决该问题; 不妨设数列A元素个数为n，数列B元素个数为m，各自升序排序，求第k小元素; 取A[k / 2] B[k / 2] 比较; 如果 A[k / 2] > B[k / 2] 那么，所求的元素必然不在B的前k / 2个元素中(证明反证法); 反之，必然不在A的前k / 2个元素中，于是我们可以将A或B数列的前k / 2元素删去，求剩下两个数列的; k - k / 2小元素，于是得到了数据规模变小的同类问题，递归解决; 如果 k / 2 大于某数列个数，所求元素必然不在另一数列的前k / 2个元素中，同上操作就好。

相关： Data Stream Median

Solution

```
class Solution {
    /**
```

```
     * @param A: An integer array.
     * @param B: An integer array.
     * @return: a double whose format is *.5 or *.0
     */
    public double findMedianSortedArrays(int[] A, int[] B) {
        int totalLength = A.length + B.length;
        if (totalLength % 2 == 1) {
            return findKth(A, 0, B, 0, totalLength / 2 + 1);
        }
        return (findKth(A, 0, B, 0, totalLength / 2) + findKth(A
, 0, B, 0, totalLength / 2 + 1)) / 2.0;
    }

    public int findKth(int[] A, int A_start, int[] B, int B_star
t, int k) {
        if (A_start >= A.length) {
            return B[B_start + k - 1];
        }
        if (B_start >= B.length) {
            return A[A_start + k - 1];
        }

        if (k == 1) {
            return Math.min(A[A_start], B[B_start]);
        }


        int A_key = A_start + k / 2 - 1 < A.length
                ? A[A_start + k / 2 - 1]
                : Integer.MAX_VALUE;
        int B_key = B_start + k / 2 - 1 < B.length
                ? B[B_start + k / 2 - 1]
                : Integer.MAX_VALUE;

        if (A_key < B_key) {
            return findKth(A, A_start + k / 2, B, B_start, k - k
 / 2);
        } else {
            return findKth(A, A_start, B, B_start + k / 2, k - k
 / 2);
```

```
            }


        }
    }
```

Intersection of Two Arrays

Question

> Given two arrays, write a function to compute their intersection.
>
> Each element in the result must be unique. The result can be in any order.
>
> Example Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2].

Analysis

非常直观的解法就是利用HashMap，循环nums1[]一遍，记录下nums1[]中出现的数字，再对nums2[]做循环，如果在HashMap中出现过，则记录到最后的结果中。

此外也可以用先sort的方法，通过merge来保证结果unique。

Solution

```
// HashMap Approach
public class Solution {
    /**
     * @param nums1 an integer array
     * @param nums2 an integer array
     * @return an integer array
     */
    public int[] intersection(int[] nums1, int[] nums2) {
        HashMap<Integer, Boolean> map1 = new HashMap<Integer, Bo
olean>();
        HashMap<Integer, Boolean> intersectMap = new HashMap<Int
eger, Boolean>();
        for (int i = 0; i < nums1.length; i++) {
            if (!map1.containsKey(nums1[i])) {
                map1.put(nums1[i], true);
            }
        }
        for (int j = 0; j < nums2.length; j++) {
            if (map1.containsKey(nums2[j]) && !intersectMap.cont
ainsKey(nums2[j])) {
                intersectMap.put(nums2[j], true);
            }
        }
        int[] result = new int[intersectMap.size()];
        int i = 0;
        for (Integer e : intersectMap.keySet()) {
            result[i] = e;
            i++;
        }
        return result;
    }
}
```

```java
// Sort & Merge
public class Solution {
    /**
     * @param nums1 an integer array
     * @param nums2 an integer array
     * @return an integer array
     */
    public int[] intersection(int[] nums1, int[] nums2) {
        Arrays.sort(nums1);
        Arrays.sort(nums2);

        int i = 0, j = 0;
        int[] temp = new int[nums1.length];
        int index = 0;
        while (i < nums1.length && j < nums2.length) {
            if (nums1[i] == nums2[j]) {
                if (index == 0 || temp[index - 1] != nums1[i]) {
                    temp[index++] = nums1[i];
                }
                i++;
                j++;
            } else if (nums1[i] < nums2[j]) {
                i++;
            } else {
                j++;
            }
        }

        int[] result = new int[index];
        for (int k = 0; k < index; k++) {
            result[k] = temp[k];
        }

        return result;
    }
}
```

Intersection of Two Arrays II

Question

> Given two arrays, write a function to compute their intersection.

Notice

Each element in the result should appear as many times as it shows in both arrays. The result can be in any order.

Example

Given nums1 = [1, 2, 2, 1], nums2 = [2, 2], return [2, 2].

Challenge

What if the given array is already sorted? How would you optimize your algorithm?

What if nums1's size is small compared to num2's size? Which algorithm is better?

What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

Analysis

本题的问题与系列中的第一题基本相同，只是需要考虑在intersection中重复的数量。同样利用HashMap记录，不过这次需要记录nums1[]中出现次数，并且在结果的resultMap中记录同时在nums2[]出现的次数。

Solution

```
public class Solution {
    /**
     * @param nums1 an integer array
     * @param nums2 an integer array
```

```
     * @return an integer array
     */
    public int[] intersection(int[] nums1, int[] nums2) {
        HashMap<Integer, Integer> map = new HashMap<Integer, Int
eger>();
        HashMap<Integer, Integer> resultMap = new HashMap<Intege
r, Integer>();
        ArrayList<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < nums1.length; i++) {
            if (!map.containsKey(nums1[i])) {
                map.put(nums1[i], 1);
            } else {
                map.put(nums1[i], map.get(nums1[i])  + 1);
            }
        }

        for (int j = 0; j < nums2.length; j++) {
            if (map.containsKey(nums2[j]) && map.get(nums2[j]) >
0) {
                map.put(nums2[j], map.get(nums2[j]) - 1);
                if (!resultMap.containsKey(nums2[j])) {
                    resultMap.put(nums2[j], 1);
                } else {
                    resultMap.put(nums2[j], resultMap.get(nums2[
j]) + 1);
                }
            }
        }

        int sum = 0;
        for (Integer e : resultMap.keySet()) {
            int count = resultMap.get(e);
            sum += count;
            for (int i = 0; i < count; i++) {
                list.add(e);
            }
        }
        int[] result = new int[sum];
        for (int i = 0; i < sum; i++) {
            result[i] = (int) list.get(i);
```

```
            }
            return result;
        }
    }
```

Maximum Subarray

http://www.lintcode.com/en/problem/maximum-subarray/

Question

> Given an array of integers, find a contiguous subarray which has the largest sum.
>
> Example
>
> Given the array [−2,2,−3,4,−1,2,1,−5,3], the contiguous subarray [4,−1,2,1] has the largest sum = 6.

Analysis

DP和Greedy均可进行求解，不过还是更推荐使用DP，因为Greedy很多情况下都是不正确的。

Solution

Compare all subarray sum

```
// Compare all subarray sum: O(n^2)
```

Dynamic Programming

```
// Dynamic Programming: O(n)
// Similar idea to Greedy Algorithm

public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(int[] A) {
        int n = A.length;
        int[] dp = new int[n]; //dp[i] means the maximum subarra
y ending with A[i];
        dp[0] = A[0];
        int max = dp[0];

        for(int i = 1; i < n; i++){
            dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
            max = Math.max(max, dp[i]);
        }

        return max;
    }
}
```

Greedy Algorithm

```java
// Greedy Algorithm: O(n)
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A integer indicate the sum of max subarray
     */
    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int length = nums.length;
        int max = Integer.MIN_VALUE;
        int sum = 0;
        for (int i = 0; i < length; i++) {
            sum = sum + nums[i];
            max = Math.max(sum, max);
            sum = Math.max(sum, 0);
        }
        return max;
    }
}
```

Reference

- programcreek: Maximum Subarray (Java)
- O(n^3) algorithm
- O(n^2) algorithm
- O(NlogN) algorithm
- O(N) algorithm
- Video Tutorial

Subarray Sum Closest

Problem

> Given an integer array, find a subarray with sum closest to zero. Return the indexes of the first number and last number.
>
> **Example**
>
> Given `[-3, 1, 1, -3, 5]`, return `[0, 2], [1, 3], [1, 1], [2, 2]` or `[0, 4]`.

Analysis

相比于Subarray Sum问题，这里同样可以记录下位置i的sum，存入一个数组或者链表中，按照sum的值sort，再寻找相邻两个sum差值绝对值最小的那个，也就得到了subarray sum closest to 0。

延伸：Subsequence with sum closest to t

Solution

```
class Pair {
    int sum;
    int index;
    public Pair(int s, int i) {
        sum = s;
        index = i;
    }
}

public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
```

```
     *          and the index of the last number
     */
    public int[] subarraySumClosest(int[] nums) {
        int[] res = new int[2];
        if (nums == null || nums.length == 0) {
            return res;
        }

        int len = nums.length;
        if(len == 1) {
            res[0] = res[1] = 0;
            return res;
        }
        Pair[] sums = new Pair[len+1];
        int prev = 0;
        sums[0] = new Pair(0, 0);
        for (int i = 1; i <= len; i++) {
            sums[i] = new Pair(prev + nums[i-1], i);
            prev = sums[i].sum;
        }
        Arrays.sort(sums, new Comparator<Pair>() {
           public int compare(Pair a, Pair b) {
               return a.sum - b.sum;
           }
        });
        int ans = Integer.MAX_VALUE;
        for (int i = 1; i <= len; i++) {

            if (ans > sums[i].sum - sums[i-1].sum) {
                ans = sums[i].sum - sums[i-1].sum;
                int[] temp = new int[]{sums[i].index - 1, sums[i
- 1].index - 1};
                Arrays.sort(temp);
                res[0] = temp[0] + 1;
                res[1] = temp[1];
            }
        }

        return res;
    }
```

```
}
```

Reference

http://www.jiuzhang.com/solutions/subarray-sum-closest/

http://rafal.io/posts/subsequence-closest-to-t.html

Subarray Sum

Question

Given an integer array, find a subarray where the sum of numbers is zero. Your code should return the index of the first number and the index of the last number.

Example

Given `[-3, 1, 2, -3, 4]`, return `[0, 2]` or `[1, 3]`.

Analysis

记录每一个位置的sum，存入HashMap中，如果某一个sum已经出现过，那么说明中间的subarray的sum为0. 时间复杂度O(n)，空间复杂度O(n)

Solution

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: A list of integers includes the index of the fir
st number
     *          and the index of the last number
     */
    public ArrayList<Integer> subarraySum(int[] nums) {
        // write your code here

        int len = nums.length;

        ArrayList<Integer> ans = new ArrayList<Integer>();
        HashMap<Integer, Integer> map = new HashMap<Integer, Int
eger>();

        map.put(0, -1);

        int sum = 0;
        for (int i = 0; i < len; i++) {
            sum += nums[i];

            if (map.containsKey(sum)) {
                ans.add(map.get(sum) + 1);
                ans.add(i);
                return ans;
            }

            map.put(sum, i);
        }

        return ans;
    }
}
```

Plus One

Question

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

**Example*8

Given [1,2,3] which represents 123, return [1,2,4].

Given [9,9,9] which represents 999, return [1,0,0,0].

Tags

Array Google

Related Problems

Medium Divide Two Integers 15 % Easy Add Binary

Analysis

比较巧的方法是将 +1 当做末位digit的进位carries；每次进位 `carries = sum / 10;` ，留下的数字为 `digits[i] = sum % 10;`

如果最终进位 ！= 0，说明要新增加一位，则新建数组把计算结果填进去。

Solution

```java
public class Solution {
    /**
     * @param digits a number represented as an array of digits
     * @return the result
     */
    public int[] plusOne(int[] digits) {
        int carries = 1;
        for (int i = digits.length - 1; i >= 0 && carries > 0; i
--) {
            int sum = digits[i] + carries;
            digits[i] = sum % 10;
            carries = sum / 10;
        }
        if (carries == 0) {
            return digits;
        }
        int[] result = new int[digits.length + 1];
        result[0] = 1;
        for (int i = 1; i < result.length; i++) {
            result[i] = digits[i - 1];
        }
        return result;
    }
}
```

Dynamic Programming

动态规划的4点要素

1 . 状态 **State**

- 灵感,创造力,存储小规模问题的结果
  - 最优解/Maximum/Minimum
  - Yes/No
  - Count

2 . 方程 **Function**

- 状态之间的联系,怎么通过小的状态,来求得大的状态

3 . 初始化 **Intialization**

- 最极限的小状态是什么, 起点

4 . 答案 **Answer**

- 最大的那个状态是什么,终点

动态规划中的优化

滚动数组优化

```
f[i] = max(f[i-1], f[i-2] + A[i]);
```

转换为

```
f[i%2] = max(f[(i-1)%2]和 f[(i-2)%2])
```

一维滚动数组优化

这类题目特点

`f[i] = max(f[i-1], f[i-2] + A[i])` ; 由 `f[i-1],f[i-2]` 来决定状态

可以转化为

`f[i%2] = max(f[(i-1)%2]` 和 `f[(i-2)%2])` 由 `f[(i-1)%2]` 和 `f[(i-2)%2]` 来决定状态

观察我们需要保留的状态来确定模数

二维动态规划空间优化

这类题目特点

`f[i][j]` = 由 `f[i-1]` 行 来决定状态, 第 `i` 行跟 `i-1` 行之前毫无关系

所以状态转变为

`f[i%2][j]` = 由 `f[(i-1)%2]` 行来决定状态

记忆化搜索

- 本质上: 动态规划
- 动态规划就是解决了重复计算的搜索
- 动态规划的实现方式:
    - 循环(从小到大递推)
    - 记忆化搜索(从大到小搜索)
        - 画搜索树
        - 万金油

什么时候用记忆化搜索?

- 状态转移特别麻烦,不是顺序性。
    - Longest Increasing continuous Subsequence 2D
        - 遍历x,y 上下左右四个格子 `dp[x][y] = dp[nx][ny]`
    - Coins in a Line III
        - `dp[i][j] = sum[i][j] - min(dp[i+1][j], dp[i][j-1])` ;
- 初始化状态不是很容易找到

    - Stone Game
        - 初始化 `dp[i][i] = 0`
    - Longest Increasing continuous Subsequence 2D
        - 初始化极小值
- 从大到小

博弈类DP

博弈有先后手

- State
    - 定义一个人的状态
- Function
    - 考虑两个人的状态做状态更新
- Initialize
- Answer

先思考最小状态，然后思考大的状态 -> 往小的状态地推，那么非常适合记忆化搜索

区间类DP

区间类DP特点:

1. 求一段区间的解max/min/count
2. 转移方程通过区间更新
3. 从大到小的更新

背包类DP

背包类DP 特点:

1. 用值作为DP维度
2. Dp过程就是填写矩阵
3. 可以滚动数组优化

参考资料

- Hawstein: 动态规划：从新手到专家
- 什么是动态规划？动态规划的意义是什么?

House Robber

Question

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Example

Given `[3, 8, 4]`, return `8`.

Challenge

O(n) time and O(1) memory.

Tags

LinkedIn Dynamic Programming Airbnb

Related Problems

Medium House Robber III 28 % Medium House Robber II 28 % Medium Paint House 35 % Easy Paint Fence 28 % Naive Fibonacci 25 % Easy Climbing Stairs

Analysis

根据 动态规划**4**要素 进行分析：

类型：序列型动态规划

- 状态 State
  - f[i] 表示前i个房子中,偷到的最大价值

- 方程 Function
  - f[i] = max(f[i-1], f[i-2] + A[i-1]);
- 初始化 Intialization
  - f[0] = 0;
  - f[1] = A[0];
- 答案 Answer
  - f[n]

状态转移方程将f[i]分了两种情况（这里定义i是第i个房子，因此数组下标要减1，使用A[i-1]）：

1. 去A[i-1]，那么f[i] = f[i-2] + A[i-1]
2. 不去A[i-1]，那么f[i] = f[i-1] 两者取最大值就是f[i]

滚动数组优化空间复杂度

因为只与前面两个dp状态有关，因此可以通过滚动数组优化，状态转移方程变为：`dp[i%2] = Math.max(dp[(i-1)%2], dp[(i-2)%2] + A[i-1]);` 优化后：Space O(1), Time O(n)

另：除了标准的4要素分析，利用O(n)空间的DP，以及通过滚动数组优化后O(1)空间的DP，还有一种更简便的利用O(1)空间的DP，基本思想是相同的，都是分解成为两种情况，rob当前的房子i，和不rob当前的房子i。不过因为只需要知道遍历完之后的最终结果，所以并不保留中间过程的变量，使得空间复杂度将为O(1)。当然空间复杂度都是O(n)，即遍历数组一遍。这种方法的本质其实正是滚动数组的优化。具体方法可以参考LeetCode讨论：https://discuss.leetcode.com/topic/11082/java-o-n-solution-space-o-1/13

Solution

DP Solution（一维序列型动态规划） with O(n) space, O(n) time

```java
public class Solution {
    /**
     * @param A: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public int houseRobber(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        // Define DP state
        int[] dp = new int[A.length + 1];

        // Initialize DP
        dp[0] = 0;
        dp[1] = A[0];

        // DP Function
        for (int i = 2; i <= A.length; i++) {
            dp[i] = Math.max(dp[i-1], dp[i-2] + A[i-1]);
        }
        return dp[A.length];
    }
}
```

DP Solution Optimized with Rolling Array （滚动数组）, with O(1) space and O(n) time

```
public class Solution {
    /**
    * @param A: An array of non-negative integers.
    * return: The maximum amount of money you can rob tonight
    */
    public int houseRobber(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }
        int []dp = new int[2];

        dp[0] = 0;
        dp[1] = A[0];

        for(int i = 2; i <= n; i++) {
            dp[i%2] = Math.max(dp[(i-1)%2], dp[(i-2)%2] + A[i-1]
);
        }
        return dp[n%2];
    }
}
```

Two Variables with O(1) space, O(n) time

```
public static int rob(int[] nums) {
    int ifRobbedPrevious = 0;     // max monney can get if rob c
urrent house
    int ifDidntRobPrevious = 0; // max money can get if not rob
current house

    // We go through all the values, we maintain two counts, 1)
if we rob this cell, 2) if we didn't rob this cell
    for(int i=0; i < nums.length; i++) {
        // If we rob current cell, previous cell shouldn't be ro
bbed. So, add the current value to previous one.
        int currRobbed = ifDidntRobPrevious + nums[i];

        // If we don't rob current cell, then the count should b
e max of the previous cell robbed and not robbed
        int currNotRobbed = Math.max(ifDidntRobPrevious, ifRobbe
dPrevious);

        // Update values for the next round
        ifDidntRobPrevious  = currNotRobbed;
        ifRobbedPrevious = currRobbed;
    }

    return Math.max(ifRobbedPrevious, ifDidntRobPrevious);
}
```

Reference

- LeetCode: House Robber Java O(n) solution, space O(1)
- ProgramCreek: LeetCode House Robber

House Robber II

Question

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are **arranged in a circle**. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Notice

This is an extension of House Robber.

Example

nums = [3,6,4], return 6

Tags

Dynamic Programming Microsoft

Related Problems

Medium House Robber III 28 % Medium Paint House 35 % Easy Paint Fence 28 % Medium House Robber

Analysis

House Robber的延伸问题，将线性（linear）排列改成环形（cycle），DP的策略需要进行相应的调整，由于定义了不能选择相邻的房子，可以分别计算两种情况，一个选择 `nums[0]` ，那么就不能选择 `nums[nums.length]` ，或者选择 `nums[nums.length]` ，就不可以选择 `nums[0]` ，这样，环形的问题就分解成为两个线性问题，最后取两个结果中的最大值即可。

简单的示例如下：

```
nums = [3,6,4]
```

第一种，选 `nums[0]`

```
[3,6,X]
```

第二种，选 `nums[nums.length]`

```
[X,6,4]
```

为了下标的标注方便，统一两个DP数组的长度，只不过在最终的统计结果时，选 `nums[0]` 取DP数组的 `first[nums.length - 1]`，而选 `nums[nums.length]` ，则取DP数组中的 `second[nums.length]` 。

另外，因为是I的延伸题，I中所运用的DP可以被复用，只需要设定起始点和终点，那么问题II，就可以直接拆解为两个不同起始点和终点的问题I。
https://discuss.leetcode.com/topic/14375/simple-ac-solution-in-java-in-o-n-with-explanation

Solution

```
public class Solution {
    /**
     * @param nums: An array of non-negative integers.
     * return: The maximum amount of money you can rob tonight
     */
    public int houseRobber2(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length < 2) {
            return nums[0];
        }

        int[] first = new int[nums.length + 1]; // Start with fi
rst house
        int[] second = new int[nums.length + 1]; // Start with s
econd house

        first[0] = 0;
        first[1] = nums[0];
        second[0] = 0;
        second[1] = 0;

        for (int i = 2; i <= nums.length; i++) {
            first[i] = Math.max(first[i - 1], first[i - 2] + num
s[i - 1]);
            second[i] = Math.max(second[i - 1], second[i - 2] +
nums[i - 1]);
        }
        return Math.max(first[nums.length - 1], second[nums.leng
th]);
    }
}
```

Utilizing House Robber I

```
public class Solution {
    /**
     * @param nums: An array of non-negative integers.
```

```
     * return: The maximum amount of money you can rob tonight
     */
    public int houseRobber2(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length < 2) {
            return nums[0];
        }

        return Math.max(houseRobber(nums, 0, nums.length - 2), h
ouseRobber(nums, 1, nums.length - 1));
    }

    public int houseRobber(int[] A, int start, int end) {
        if (A == null || A.length == 0) {
            return 0;
        }

        if (start == end) {
            return A[start];
        }
        if (start + 1 == end) {
            return Math.max(A[start], A[end]);
        }

        // Define DP state
        int[] dp = new int[end - start + 2];

        // Initialize DP
        dp[start] = A[start];
        dp[start + 1] = Math.max(A[start], A[start + 1]);

        // DP Function
        for (int i = start + 2; i <= end; i++) {
            dp[i] = Math.max(dp[i-1], dp[i-2] + A[i]);
        }
        return dp[end];
    }
}
```

Utilize House Robber I with Rolling Array Optimization

```
public class Solution {
    public int houseRobber2(int[] nums) {
        if (nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }
        return Math.max(houseRobber1(nums, 0, nums.length - 2),
houseRobber1(nums, 1, nums.length - 1));
    }

    public int houseRobber1(int[] nums, int st, int ed) {
        int []res = new int[2];
        if(st == ed)
            return nums[ed];
        if(st+1 == ed)
            return Math.max(nums[st], nums[ed]);
        res[st%2] = nums[st];
        res[(st+1)%2] = Math.max(nums[st], nums[st+1]);

        for(int i = st+2; i <= ed; i++) {
            res[i%2] = Math.max(res[(i-1)%2], res[(i-2)%2] + num
s[i]);

        }
        return res[ed%2];
    }
}
```

Reference

- LeetCode discuss: Simple AC solution in Java in O(n) with explanation
- Jiuzhang

Longest Increasing Continuous Subsequence

Question

Give an integer array，find the longest increasing continuous subsequence in this array.

An increasing continuous subsequence:

Can be from right to left or from left to right. Indices of the integers in the subsequence should be continuous.

Notice

O(n) time and O(1) extra space.

Example

For [5, 4, 2, 1, 3], the LICS is [5, 4, 2, 1], return 4.

For [5, 1, 2, 3, 4], the LICS is [1, 2, 3, 4], return 4.

Tags

Enumeration Dynamic Programming Array

Related Problems

Hard Longest Increasing Continuous subsequence II

Analysis

很直观的思路就是进行两次遍历，第一次从左到右，第二次从右往左，总是检查当前序列是否单调递增，并且记录当前最长的上升序列长度。

Check out the follow up question: Longest Increasing Continuous subsequence II

Solution

```
public class Solution {
    /**
     * @param A an array of Integer
     * @return  an integer
     */
    public int longestIncreasingContinuousSubsequence(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int ans = 1;
        int N = A.length;

        // From left to right
        int len = 1;
        for (int i = 1; i < N; i++) {
            if (A[i] > A[i - 1]) {
                len++;
            } else {
                len = 1;
            }
            ans = Math.max(len, ans);
        }

        // From right to left
        len = 1;
        for (int i = N - 1; i > 0; i--) {
            if (A[i - 1] > A[i]) {
                len++;
            } else {
                len = 1;
            }
            ans = Math.max(len, ans);
        }

        return ans;
```

```
        }
    }
```

Longest Increasing Continuous subsequence II

Question

Give you an integer matrix (with row size n, column size m)，find the longest increasing continuous subsequence in this matrix. (The definition of the longest increasing continuous subsequence here can start at any row or column and go up/down/right/left any direction).

Example

Given a matrix:

```
[
 [1 ,2 ,3 ,4 ,5],
 [16,17,24,23,6],
 [15,18,25,22,7],
 [14,19,20,21,8],
 [13,12,11,10,9]
]
```

return 25

Challenge

O(nm) time and memory.

Tags

Dynamic Programming

Related Problems

Easy Longest Increasing Continuous Subsequence

Analysis

可以借鉴在I问题中的思路，来构建2D的状态转移方程。

在2D中搜索increasing continuous subsequence, 可以考虑双重循环加递归搜索，直接求以 `(x, y)` 为结尾的最长子序列。

利用记忆化搜索可以进行优化。

什么时候用记忆化搜索?

1. 状态转移特别麻烦，不是顺序性。
2. 初始化状态不是很容易找到。

动态规划求解四要素：

- 状态
  - `dp[x][y]` ：以 `x, y` 作为结尾的最长子序列
- 方程
  - 遍历 `x, y` 上下左右四个方向的格子
  - `dp[x][y] = dp[nx][ny] + 1` if `A[x][y] > A[nx][ny]`
- 初始化
  - `dp[x][y]` 是极小值时，初始化为1
- 答案
  - `d[x][y]` 中的最大值

Solution

```
public class Solution {
    int[][] dp;
    int[][] flag;
    int M, N;

    int[] dx = new int[] {-1, 0, 1, 0};
    int[] dy = new int[] {0, -1, 0, 1};
    /**
     * @param A an integer matrix
     * @return  an integer
```

```
     */
    public int longestIncreasingContinuousSubsequenceII(int[][]
A) {
        if (A == null || A.length == 0 || A[0].length == 0) {
            return 0;
        }
        M = A.length;
        N = A[0].length;
        dp = new int[M][N];
        flag = new int[M][N];

        int ans = 0;

        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                dp[i][j] = search(i, j, A);
                ans = Math.max(ans, dp[i][j]);
            }
        }

        return ans;
    }

    // Memorized search, recursive
    public int search(int x, int y, int[][] A) {
        if (flag[x][y] != 0) {
            return dp[x][y];
        }

        int ans = 1; // Initialize dp[x][y] = 1 if it's local mi
n compare to 4 directions
        int nx, ny;
        for (int i = 0; i < dx.length; i++) {
            nx = x + dx[i];
            ny = y + dy[i];
            if (insideBoundary(nx, ny) && (A[x][y] > A[nx][ny]))
 {
                ans = Math.max(ans, search(nx, ny, A) + 1);
            }
        }
```

```
            flag[x][y] = 1;
            dp[x][y] = ans;

            return ans;
        }

        public boolean insideBoundary(int x, int y) {
            return (x >=0 && x < M && y >= 0 && y < N);
        }
    }
```

Reference

- Jiuzhang: Longest Increasing Continuous subsequence II

Coins in a Line

Question

There are n coins in a line. Two players take turns to take one or two coins from right side until there are no more coins left. The player who take the last coin wins.

Could you please decide the first play will win or lose?

Example

n = 1, return true.

n = 2, return true.

n = 3, return false.

n = 4, return true.

n = 5, return true.

Challenge

O(n) time and O(1) memory

Tags

Greedy Dynamic Programming Array Game Theory

Related Problems

Hard Coins in a Line III 30 % Medium Coins in a Line II

Analysis

Dynamic Programming

这一个问题可以归类到博弈类问题，需要注意的是博弈有先后手。

- State:

    - 定义一个人的状态: dp[i], 现在还剩i个硬币，现在当前取硬币的人最后输赢状况
- Function:
    - 考虑两个人的状态做状态更新: dp[i] = (!dp[i-1]) || (!dp[i-2])
- Intialize:
    - dp[0] = false
    - dp[1] = true
    - dp[2] = true
- Answer:
    - dp[n]

先思考最小状态 然后思考大的状态 -> 往小的递推，那么非常适合记忆化搜索

Algorithm with O(1) Time, O(1) Space

其实此问题如果从另一个角度思考，就是从最后剩余1个或2个硬币时进行倒推，寻找规律：

先手输：

```
o o o | o o o
```

先手胜：

```
o | o o o
```

制胜的方法就是一定在倒数第二个回合时，让对手面对3个硬币，这样因为自己可以拿1或者2个硬币，那么无论对手选1个或者2个，己方都可以拿到最后一个硬币。这个规律就是每次让对手都面对3的倍数个硬币，那么无论对方取1个或者2个，只需要取相应的硬币数，让剩下的硬币数目保持3X，这样就能够保证取胜。对于先手而言，如果自己第一轮面对的就是3的倍数个硬币，那么对手则可以使用同样的策略让自己一方每次面对3X个硬币。于是先手是否获胜的唯一要素就是初始硬币数目，在不为3的整数倍情况下，先手都可以获胜。这样的话，算法时间复杂度和空间复杂度都为 `O(1)` 。

Solution

StackOverflow

```
public class Solution {
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int n) {
        boolean[] dp = new boolean[n + 1];
        boolean[] flag = new boolean[n + 1];
        return search(n, dp, flag);
    }

    boolean search(int i, boolean[] dp, boolean[] flag) {
        if (flag[i] == true) {
            return dp[i];
        }
        if (i == 0) {
            dp[i] = false;
        } else if (i == 1) {
            dp[i] = true;
        } else if (i == 2) {
            dp[i] = true;
        } else {
            dp[i] = ! (search(i - 1, dp, flag) && search(i - 2,
dp, flag));
        }
        flag[i] = true;
        return dp[i];
    }
}
```

Improved

```
public class Solution {
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
```

```
er will win
     */
    public boolean firstWillWin(int n) {
        // write your code here
        int []dp = new int[n+1];

        return MemorySearch(n, dp);

    }
    boolean MemorySearch(int n, int []dp) { // 0 is empty, 1 is
false, 2 is true
        if(dp[n] != 0) {
            if(dp[n] == 1)
                return false;
            else
                return true;
        }
        if(n <= 0) {
            dp[n] = 1;
        } else if(n == 1) {
            dp[n] = 2;
        } else if(n == 2) {
            dp[n] = 2;
        } else if(n == 3) {
            dp[n] = 1;
        } else {
            if((MemorySearch(n-2, dp) && MemorySearch(n-3, dp))
||
                (MemorySearch(n-3, dp) && MemorySearch(n-4, dp)
)) {
                dp[n] = 2;
            } else {
                dp[n] = 1;
            }
        }
        if(dp[n] == 2)
            return true;
        return false;
    }
}
```

N % 3 - Time O(1), Space O(1)

```
public class Solution {
    /**
     * @param n: an integer
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int n) {
        if (n % 3 == 0) {
            return false;
        }
        return true;
    }
}
```

Reference

- Jiuzhang

Coins in a Line II

Question

There are n coins with different value in a line. Two players take turns to take one or two coins from left side until there are no more coins left. The player who take the coins with the most value wins.

Could you please decide the **first** player will win or lose?

Example

Given values array A = [1,2,2], return true.

Given A = [1,2,4], return false.

Tags

Dynamic Programming Array Game Theory

Related Problems

Hard Coins in a Line III 30 % Medium Coins in a Line

Analysis

动态规划4要素

- State:
  - dp[i] 现在还剩i个硬币，现在当前取硬币的人最后最多取硬币价值
- Function:
  - i 是所有硬币数目
  - sum[i] 是后i个硬币的总和
  - dp[i] = sum[i]-min(dp[i-1], dp[i-2])
- Intialize:
  - dp[0] = 0
  - dp[1] = coin[n-1]
  - dp[2] = coin[n-2] + coin[n-1]

- Answer:
  - dp[n]

可以画一个树形图来解释：

```
                    [5, 1, 2, 10] dp[4]
          (take 5) /              \ (take 5, 1)
                  /                \
          [1, 2, 10] dp[3]          [2, 10] dp[2]
(take 1) /      \ (take 1, 2)  (take 2) / \ (take 2, 10)
        /        \                      /   \
  [2, 10] dp[2]  [10] dp[1]     [10] dp[1] [] dp[0]
```

也就是说，每次的剩余硬币价值最多值dp[i]，是当前所有剩余i个硬币价值之和sum[i]，减去下一手时对手所能拿到最多的硬币的价值，即 `dp[i] = sum[i] - min(dp[i - 1], dp[i - 2])`

Solution

```
public class Solution {
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        if (values == null || values.length == 0) {
            return false;
        }
        int n = values.length;
        int[] dp = new int[n + 1];
        boolean[] flag = new boolean[n + 1];

        int[] sum = new int[n + 1];
        int total = 0;
        sum[0] = 0;
        for (int i = n - 1; i >= 0; i--) {
            sum[n - i] = sum[n - i - 1] + values[i];
```

```
            total += values[i];
        }

        return search(n, n, dp, flag, values, sum) > total / 2;
    }
    public int search(int i, int n, int[] dp, boolean[] flag, int
[] values, int[] sum) {
        if (flag[i] == true) {
            return dp[i];
        }
        if (i == 0) {
            dp[i] = 0;
        } else if (i == 1) {
            dp[i] = values[n - 1];
        } else  if (i == 2) {
            dp[i] = values[n - 1] + values[n - 2];
        } else {
            dp[i] = sum[i] - Math.min(search(i - 1, n, dp, flag,
 values, sum), search(i - 2, n, dp, flag, values, sum));
        }
        flag[i] = true;
        return dp[i];
    }
}
```

Reference

- LeetCode Article: Coins in Line
- Dynamic Programming — Coin In a Line Game Problem
- geeksforgeeks: Dynamic Programming | Set 31

Coins in a Line III

Question

There are n coins in a line. Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

Could you please decide the first player will win or lose?

Example

Given array A = [3,2,2], return true.

Given array A = [1,2,4], return true.

Given array A = [1,20,4], return false.

Challenge

Follow Up Question:

If n is even. Is there any hacky algorithm that can decide whether first player will win or lose in O(1) memory and O(n) time?

Tags

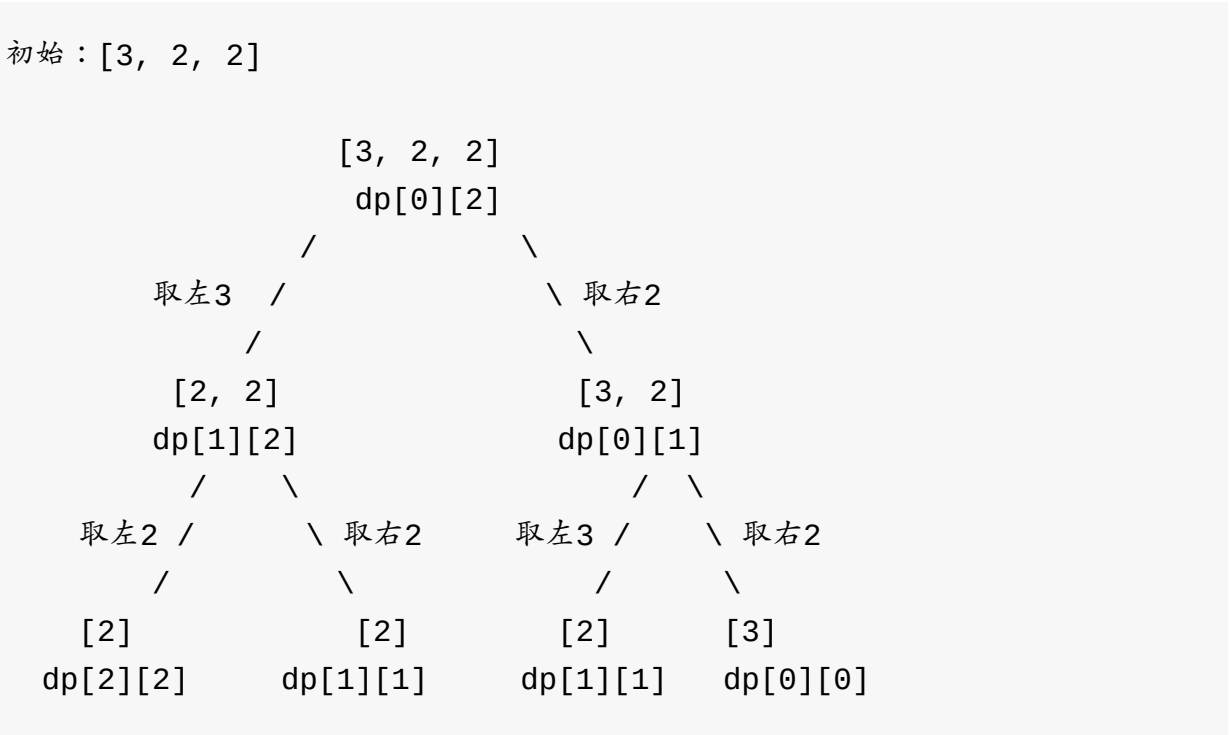
Dynamic Programming Array Game Theory

Related Problems

Medium Coins in a Line II 31 % Medium Coins in a Line

Analysis

类似于系列问题II，可以根据取硬币的流程，画出一个tree:



DP四要素：

- State:
  - `dp[i][j]` 现在还第i到第j的硬币，现在当前取硬币的人（先手）最后最多取硬币价值；这里是区间型DP，下标表示区间范围
- Function:
  - `sum[i][j]` 第i到第j的硬币价值总和
  - `dp[i][j] = sum[i][j] - min(dp[i+1][j], dp[i][j-1]);`
- Initialize:
  - `dp[i][i] = coin[i]`
- Answer:
  - `dp[0][n-1]`

运用Memorized Search，记忆化搜索可以优化时间。

Solution

```
public class Solution {
```

```
    /**
     * @param values: an array of integers
     * @return: a boolean which equals to true if the first play
er will win
     */
    public boolean firstWillWin(int[] values) {
        if (values == null || values.length == 0) {
            return false;
        }

        int n = values.length;

        int[][] sum = new int[n+1][n+1];
        int[][] dp = new int[n+1][n+1];
        boolean[][] flag = new boolean[n+1][n+1];

        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                if (i == j) {
                    sum[i][j] = values[j];
                } else {
                    sum[i][j] = sum[i][j - 1] + values[j];
                }
            }
        }
        int total = 0;
        for (int i = 0; i < n; i++) {
            total += values[i];
        }

        return (total / 2) < search(0, n - 1, dp, flag, values,
sum);
    }

    public int search(int i, int j, int[][] dp, boolean[][] flag
, int[] values, int[][] sum) {
        if (flag[i][j]) {
            return dp[i][j];
        }
```

```
            flag[i][j] = true;

            if (i == j) {
                dp[i][j] = values[i];
            } else if (i > j) {
                dp[i][j] = 0;
            } else if (i + 1 == j) {
                dp[i][j] = Math.max(values[i], values[j]);
            } else {
                dp[i][j] = sum[i][j] - Math.min(search(i, j - 1, dp,
 flag, values, sum), search(i + 1, j, dp, flag, values, sum));
            }

            return dp[i][j];
        }

}
```

Reference

- Jiuzhang: Coins in a Line III

Maximum Product Subarray

Question

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

Example For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

Tags

LinkedIn Dynamic Programming Subarray

Related Problems

Medium Best Time to Buy and Sell Stock 41 % Medium Maximum Subarray Difference 23 % Easy Minimum Subarray 38 % Medium Maximum Subarray II

Analysis

本题其实是Maximum Subarray Sum问题的延伸，不同的是这里需要考虑元素的符号。

DP的四要素

- 状态：
  - `max_product[i]`：以nums[i]结尾的max subarray product
  - `min_product[i]`：以nums[i]结尾的min subarray product
- 方程：
  - `max_product[i] = getMax(max_product[i-1] * nums[i], min_product[i-1] * nums[i], nums[i])`
  - `min_product[i] = getMin(max_product[i-1] * nums[i], min_product[i-1] * nums[i], nums[i])`
- 初始化：

    - `max_product[0] = min_product[0] = nums[0]`
- 结果：
    - 每次循环中 max_product[i] 的最大值

Solution

DP, with O(n) space, O(n) time

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: an integer
     */
    public int maxProduct(int[] nums) {

        int[] max = new int[nums.length]; // maximum product end
ing with nums[i]
        int[] min = new int[nums.length]; // minimum product end
ing with nums[i]

        max[0] = min[0] = nums[0];

        int result = max[0];

        for (int i = 1; i < nums.length; i++) {
            max[i] = min[i] = nums[i];
            if (nums[i] > 0) { // 区分nums[i]的符号
                max[i] = Math.max(max[i], nums[i] * max[i - 1]);
                min[i] = Math.min(min[i], nums[i] * min[i - 1]);
            } else {
                max[i] = Math.max(max[i], nums[i] * min[i - 1]);
                min[i] = Math.min(min[i], nums[i] * max[i - 1]);
            }
            result = Math.max(max[i], result);
        }

        return result;
    }
}
```

DP, improved, with O(1) space, O(n) time

```
public class Solution {
    /**
     * @param nums: an array of integers
     * @return: an integer
     */
    public int maxProduct(int[] nums) {
        int result, currentMax, currentMin;
        result = currentMax = currentMin = nums[0];

        for (int i = 1; i < nums.length; i++) {
            int temp = currentMax;
            currentMax = Math.max(nums[i], Math.max(currentMax *
 nums[i], currentMin * nums[i]));
            currentMin = Math.min(nums[i], Math.min(temp * nums[
i], currentMin * nums[i]));

            result = Math.max(currentMax, result);
        }

        return result;
    }
}
```

Reference

- programcreek: Maximum Product Subarray
- Jiuzhang
- LeeCode Discussion

Longest Palindromic Substring

Question

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

Example

Given the string = `"abcdzdcab"`, return `"cdzdc"`.

Challenge

O(n2) time is acceptable. Can you do it in O(n) time.

Tags

String

Related Problems

Easy Valid Palindrome 22 % Medium Longest Palindromic Substring 26 % Medium Palindrome Partitioning II 22 %

Analysis

区间类动态规划

Time O(n^2), Space O(n^2)

用 `dp[i][j]` 来存DP的状态，需要较多的额外空间: Space O(n^2)

DP的4个要素

- 状态：
  - `dp[i][j]` : s.charAt(i)到s.charAt(j)是否构成一个Palindrome
- 转移方程：

    - `dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i + 1][j - 1])`
- 初始化：
    - `dp[i][j] = true` when `j - i <= 2`
- 结果：
    - 找 `maxLen = j - i + 1;`，并得到相应longest substring： `longest = s.substring(i, j + 1);`

中心扩展

这种方法基本思想是遍历数组，以其中的1个元素或者2个元素作为palindrome的中心，通过辅助函数，寻找能拓展得到的最长子字符串。外层循环 O(n)，内层循环 O(n)，因此时间复杂度 Time O(n^2)，相比动态规划二维数组存状态的方法，因为只需要存最长palindrome子字符串本身，这里空间更优化：Space O(1)

Manacher's Algorithm

这种算法可以达到O(n)时间，但是并不很通用，因此这里略过讨论。具体参考：

- wikipedia: Longest palindromic substring
- Manacher’s Algorithm – Linear Time Longest Palindromic Substring – Part 1

Solution

区间DP，Time O(n^2) Space O(n^2)

```
public class Solution {
    /**
     * @param s input string
     * @return the longest palindromic substring
     */
     public String longestPalindrome(String s) {
         if(s == null || s.length() <= 1) {
             return s;
         }

         int len = s.length();
         int maxLen = 1;
         boolean [][] dp = new boolean[len][len];

         String longest = null;
         for(int k = 0; k < s.length(); k++){
             for(int i = 0; i < len - k; i++){
                 int j = i + k;
                 if(s.charAt(i) == s.charAt(j) && (j - i <= 2 ||
 dp[i + 1][j - 1])){
                     dp[i][j] = true;

                     if(j - i + 1 > maxLen){
                        maxLen = j - i + 1;
                        longest = s.substring(i, j + 1);
                     }
                 }
             }
         }

         return longest;
     }
}
```

Time O(n^2) Space O(1)

```
public class Solution {
    /**
     * @param s input string
```

```
     * @return the longest palindromic substring
     */
    public String longestPalindrome(String s) {
            if (s.isEmpty()) {
                return null;
            }

            if (s.length() == 1) {
                return s;
            }

            String longest = s.substring(0, 1);
            for (int i = 0; i < s.length(); i++) {
                // get longest palindrome with center of i
                String tmp = helper(s, i, i);
                if (tmp.length() > longest.length()) {
                    longest = tmp;
                }

                // get longest palindrome with center of i, i+1
                tmp = helper(s, i, i + 1);
                if (tmp.length() > longest.length()) {
                    longest = tmp;
                }
            }

            return longest;
    }

    // Given a center, either one letter or two letter,
    // Find longest palindrome
    public String helper(String s, int begin, int end) {
        while (begin >= 0 && end <= s.length() - 1 && s.charAt(b
egin) == s.charAt(end)) {
            begin--;
            end++;
        }
        return s.substring(begin + 1, end);
    }
}
```

Reference

- LeetCode Discussion
- *programcreek: Longest Palindromic Substring (Java)
- 水中的鱼: Longest Palindromic Substring 解题报告

Maximal Square

Question

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

Example

For example, given the following matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Return 4.

Tags

Dynamic Programming Airbnb Facebook

Related Problems

Hard Maximal Rectangle

Analysis

考虑以 `f[i][j]` 为右下角顶点可以拓展的正方形边长，那么可以由 `f[i-1][j]` ， `f[i][j-1]` ， `f[i-1][j-1]` 三者的最小值决定 `f[i][j]` 的最长边长。

> 当我们判断以某个点为正方形右下角时最大的正方形时，那它的上方，左方和左上方三个点也一定是某个正方形的右下角，否则该点为右下角的正方形最大就是它自己了。这是定性的判断，那具体的最大正方形边长呢？我们知道，该点为右下角的正方形的最大边长，最多比它的上方，左方和左上方为右下角的正方形的边长多1，最好的情况是是它的上方，左方和左上方为右下角的正方形的大小都一样的，这样加上该点就可以构成一个更大的正方形。但如果它的上方，左方和左上方为右下角的正方形的大小不一样，合起来就会缺了某个角落，这时候只能取那三个正方形中最小的正方形的边长加1了。假设 `dp[i][j]` 表示以i,j为右下角的正方形的最大边长，则有
>
> ```
> dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1
> ```
>
> 当然，如果这个点在原矩阵中本身就是0的话，那dpi肯定就是0了。

时间O(mn)， 空间O(mn)

- 状态 State
  - `f[i][j]` 表示以i和j作为正方形右下角可以拓展的最大边长
- 方程 Function
  - if `matrix[i][j] == 1`
    - `f[i][j] = min(f[i - 1][j], f[i][j-1], f[i-1][j-1]) + 1;`
  - if `matrix[i][j] == 0`
    - `f[i][j] = 0`
- 初始化 Intialization
  - `f[i][0] = matrix[i][0];`
  - `f[0][j] = matrix[0][j];`
- 答案 Answer
  - `max{f[i][j]}`

可以使用二维滚动数组优化:

1. 状态 State
   - f[i][j] 表示以i和j作为正方形右下角可以拓展的最大边长
2. 方程 Function
   - if matrix[i][j] == 1

        - f[i%2][j] = min(f[(i - 1)%2][j], f[i%2][j-1], f[(i-1)%2][j-1]) + 1;
    - if matrix[i][j] == 0
    - f[i%2][j] = 0
3. 初始化 Intialization
    - f[i%2][0] = matrix[i][0];
    - f[0][j] = matrix[0][j];
4. 答案 Answer
    - max{f[i%2][j]}

Solution

```
public class Solution {
    /**
     * @param matrix: a matrix of 0 and 1
     * @return: an integer
     */
    public int maxSquare(int[][] matrix) {
        if (matrix.length == 0) return 0;

        int m = matrix.length, n = matrix[0].length;
        int max = 0;
        int[][] dp = new int[m][n];

        // 第一列赋值
        for (int i = 0; i < m; i++) {
            dp[i][0] = matrix[i][0];
            max = Math.max(max, dp[i][0]);
        }

        // 第一行赋值
        for (int i = 0; i < n; i++) {
            dp[0][i] = matrix[0][i];
            max = Math.max(max, dp[0][i]);
        }

        // 递推
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = matrix[i][j] == 1 ? Math.min(dp[i - 1
][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1])) + 1 : 0;

                max = Math.max(max, dp[i][j]);
            }
        }
        return max * max;
    }
}
```

Reference

- segmentfault: Maximal Square 最大正方形

Stone Game

Question

There is a stone game.At the beginning of the game the player picks `n` piles of stones in a line.

The goal is to merge the stones in one pile observing the following rules:

1. At each step of the game, the player can merge two adjacent piles to a new pile.
2. The score is the number of stones in the new pile.

You are to determine the **minimum** of the total score.

Example

For `[4, 1, 1, 4]`, in the best solution, the total score is `18`:

```
1. Merge second and third piles => [4, 2, 4], score +2
2. Merge the first two piles => [6, 4]，score +6
3. Merge the last two piles => [10], score +10
```

Other two examples:

`[1, 1, 1, 1]` return `8` `[4, 4, 5, 9]` return `43`

Tags

Dynamic Programming

Related Problems

Hard Burst Balloons 29 %

Analysis

死胡同:容易想到的一个思路从小往大，枚举第一次合并是在哪? 转而用记忆化搜索的思路，从大到小，先考虑最后的 `0 ~ n-1` 合并的总花费。

DP四要素

- State:
  - `dp[i][j]` 表示把第i到第j个石子合并到一起的最小花费
- Function:
  - 预处理 `sum[i,j]` 表示i到j所有石子价值和
  - `dp[i][j] = min(dp[i][k]+dp[k+1][j]+sum[i,j])` 对于所有 `k` 属于 `{i,j}`
- Intialize:
  - for each i
    - `dp[i][i] = 0`
- Answer:
  - `dp[0][n-1]`

区间型DP，利用二维数组下标表示下标范围。 需要注意的是对状态转移方程的理解，也就是对每一种分割方式进行遍历，

Solution

```
public class Solution {

    /**
     * @param A an integer array
     * @return an integer
     */
    int search(int l, int r, int[][] f, int[][] visit, int[][] s
um) {

        if(visit[l][r] == 1) {
            return f[l][r];
        }

        if(l == r) {
            visit[l][r] = 1;
            return f[l][r];
```

```
        }

        f[l][r] = Integer.MAX_VALUE;
        for (int k = l; k < r; k++) {
            f[l][r] = Math.min(f[l][r], search(l, k, f, visit, s
um) + search(k + 1, r, f, visit, sum) + sum[l][r]);
        }

        visit[l][r] = 1;
        return f[l][r];
    }

    public int stoneGame(int[] A) {
        if (A == null || A.length == 0) {
            return 0;
        }

        int n = A.length;

        // initialize
        int[][] f = new int[n][n];
        int[][] visit = new int[n][n];

        for (int i = 0; i < n; i++) {
            f[i][i] = 0;
        }

        // preparation
        int[][] sum = new int[n][n];
        for (int i = 0; i < n; i++) {
            sum[i][i] = A[i];
            for (int j = i + 1; j < n; j++) {
                sum[i][j] = sum[i][j - 1] + A[j];
            }
        }

        return search(0, n-1, f, visit, sum);
    }
}
```

Reference

- Jiuzhang: Stone Game

Knapsack Problems

Reference

- 背包问题九讲 2.0 by Tianyi Cui

Backpack

Question

Given `n` items with size `Ai`, an integer `m` denotes the size of a backpack. How full you can fill this backpack?

Notice

You can not divide any item into small pieces.

Example

If we have 4 items with size [2, 3, 5, 7], the backpack size is 11, we can select [2, 3, 5], so that the max size we can fill this backpack is 10. If the backpack size is 12. we can select [2, 3, 7] so that we can fulfill the backpack.

You function should return the max size we can fill in the given backpack.

Challenge

O(n x m) time and O(m) memory.

O(n x m) memory is also acceptable if you do not know how to optimize memory.

Tags

LintCode Copyright Dynamic Programming Backpack

Related Problems

Medium Backpack VI 30 % Medium Backpack V 38 % Medium Backpack IV 36 % Hard Backpack III 50 % Medium Backpack II 37 %

Analysis

背包问题序列I

动态规划4要素

- State:
  - `f[i][S]` “前i”个物品，取出一些能否组成和为S
- Function:
  - `f[i][S] = f[i-1][S - a[i]]` or `f[i-1][S]`
- Initialize:
  - `f[i][0] = true`；`f[0][1..target] = false`
- Answer:
  - 检查所有的 `f[n][j]`

O(n * S)，滚动数组优化

Solution

2D with O(n *m) time and O(n* m) space.

```
public class Solution {
    /**
    * @param m: An integer m denotes the size of a backpack
    * @param A: Given n items with size A[i]
    * @return: The maximum size
    */
    public int backPack(int m, int[] A) {
        boolean[][] dp = new boolean[A.length + 1][m + 1];

        dp[0][0] = true;

        for (int i = 1; i <= A.length; i++) {
            for (int j = 0; j <= m; j++) {
                dp[i][j] = dp[i - 1][j] || (j - A[i - 1] >= 0 &&
 dp[i - 1][j - A[i - 1]]);
            }
        }

        for (int j = m; j >= 0; j--) {
            if (dp[A.length][j]) {
                return j;
            }
        }

        return 0;
    }
}
```

1D version with O(n * m) time and O(m) memory.

```
public class Solution {
    /**
    * @param m: An integer m denotes the size of a backpack
    * @param A: Given n items with size A[i]
    * @return: The maximum size
    */
    public int backPack(int m, int[] A) {
        if (A.length == 0) return 0;

        int n = A.length;
        boolean[] dp = new boolean[m + 1];
        Arrays.fill(dp, false);
        dp[0] = true;

        for (int i = 1; i <= n; i++) {
            for (int j = m; j >= 0; j--) {
                if (j - A[i - 1] >= 0 && dp[j - A[i - 1]]) {
                    dp[j] = dp[j - A[i - 1]];
                }
            }
        }


        for (int i = m; i >= 0;i--) {
            if (dp[i]) return i;
        }


        return 0;
    }
}
```

Reference

- Backpack - neverlandly - 博客园

Sqrt(x)

Question

> Implement int sqrt(int x).
>
> Compute and return the square root of x.

Example

sqrt(3) = 1

sqrt(4) = 2

sqrt(5) = 2

sqrt(10) = 3

Analysis

问题可以转化为从 1 ~ x寻找目标数字，于是二分法就可以提供O(log(n))的时间复杂度，O(1)的空间复杂度。

此外，可以考虑使用牛顿法:

```
x_(n+1) = x_n - f(x_(n))/f'(x_(n))
```

其中 S 满足 x = sqrt(S), `f(x_(n)) = (x_(n)) ^ 2 - S`

于是 `x_(n+1) = x_(n) - ((x_(n))^ - S) / (2 * x_(n)) = 1/2 * (x_(n) + S / x_(n))`

这样 只需迭代 `x_(n+1) = 1/2 * (x_(n) + S/x_(n))` ,

直到 `| x_(n+1) - x_(n) | < 1`

Solution

Binary Serach

```
class Solution {
    /**
     * @param x: An integer
     * @return: The sqrt of x
     */
    public int mySqrt(int x) {
        if (x <= 0) {
            return 0;
        }

        long start = 1;
        long end = x;

        while (start + 1 < end) {
            long mid = start + (end - start) / 2;
            if (mid == x / mid) {
                return (int) mid;
            } else if (mid > x / mid) {
                end = mid;
            } else {
                start = mid;
            }
        }

        return (int) start;
    }
}
```

Newton's Method

```
class Solution {
    /**
     * @param x: An integer
     * @return: The sqrt of x
     */
    public int mySqrt(int x) {
        if (x <= 0) {
            return 0;
        }
        if (x <= 3) {
            return 1;
        }
        long r = x;
        while (r > x / r) {
            r = (r + x/r) / 2;
        }
        return (int) r;
    }
}
```

Reference

- Newton's Method
- wikipedia: Integer square root
- wikipedia: Methods of computing square roots

Ugly Number

Question

Write a program to check whether a given number is an `ugly number`.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Notice

Note that 1 is typically treated as an ugly number.

Example

Given num = 8 return true Given num = 14 return false

Analysis

基本思路就是，对于num，如果取余数（2，3，5中其一）为0，说明对应的prime number能够被整除，所以一直除下去。最终如果是Ugly Number，该结果应当是1。时间复杂度是O(logn)。

Solution

```java
public class Solution {
    /**
     * @param num an integer
     * @return true if num is an ugly number or false
     */
    public boolean isUgly(int num) {
        if (num == 0) {
            return false;
        }
        if (num == 1) {
            return true;
        }
        int[] primes = new int[]{2, 3, 5};
        for (int i = 0; i < primes.length; i++) {
            while (num % primes[i] == 0) {
                num = num / primes[i];
            }
        }
        return num == 1;
    }
}
```

High Frequency

K Sum 系列问题总结

- Summary for LeetCode 2Sum, 3Sum, 4Sum, K Sum
- Leetcode Problem 16-20 K-sum专题
- LeetCode总结 -- kSum篇

Two Sum Closest

Question

> Given an array `nums` of n integers, find two integers in `nums` such that the sum is closest to a given number, `target` .
>
> Return the difference between the sum of the two integers and the target.

Example

Given array `nums = [-1, 2, 1, -4]` , and `target = 4` .

The minimum difference is `1` . (4 - (2 + 1) = 1).

Analysis

与3 sum closest问题相似，通过先对数组排序，再用两个指针的方法，可以满足O(nlogn) + O(n) ~ O(nlogn)的时间复杂度的要求

不同的是这里要返回的是diff，所以只用记录minDiif即可。

Solution

```java
public class Solution {
    /**
     * @param nums an integer array
     * @param target an integer
     * @return the difference between the sum and the target
     */
    public int twoSumCloset(int[] nums, int target) {
        if (nums == null || nums.length < 2) {
            return -1;
        }

        if (nums.length == 2) {
            return target - nums[0] - nums[1];
        }
        Arrays.sort(nums);
        int pl = 0;
        int pr = nums.length - 1;

        int minDiff = Integer.MAX_VALUE;
        while (pl < pr) {
            int sum = nums[pl] + nums[pr];
            int diff = Math.abs(sum - target);
            if (diff == 0) {
                return 0;
            }
            if (diff < minDiff ) {
                minDiff = diff;
            }
            if (sum > target) {
                pr--;
            } else {
                pl++;
            }
        }
        return minDiff;
    }
}
```

3 Sum

LintCode 3 Sum

Question

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Notice

Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c)

The solution set must not contain duplicate triplets.

Analysis

考虑non-descending，先sort数组，其次考虑去除duplicate； 类似2sum，可以利用two pointers，不断移动left 和 right 指针，直到找到目标，或者两指针相遇

Solution

```
public class Solution {
    /**
     * @param numbers : Give an array numbers of n integer
     * @return : Find all unique triplets in the array which giv
es the sum of zero.
     */
    public ArrayList<ArrayList<Integer>> threeSum(int[] numbers)
 {
        ArrayList<ArrayList<Integer>> result = new ArrayList<Arr
ayList<Integer>>();
        if (numbers == null || numbers.length < 3) {
            return result;
        }
        Arrays.sort(numbers);
```

```
        for (int i = 0; i < numbers.length - 2; i++) {
            if (i > 0 && numbers[i] == numbers[i - 1]) {
                continue;
            }
            int left = i + 1;
            int right = numbers.length - 1;

            while (left < right) {
                int sum = numbers[i] + numbers[left] + numbers[r
ight];
                if (sum == 0) {
                    ArrayList<Integer> tmp = new ArrayList<Integ
er>();
                    tmp.add(numbers[i]);
                    tmp.add(numbers[left]);
                    tmp.add(numbers[right]);
                    result.add(tmp);
                    left++;
                    right--;
                    while (left < right && numbers[left] == numb
ers[left - 1]) {
                        left++;
                    }
                    while (left < right && numbers[right] == num
bers[right + 1]) {
                        right--;
                    }
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }

            }
        }
        return result;
    }
}
```

Reference

- 九章算法题解 3 Sum
- MIT Paper Ilya Baran: Subquadratic Algorithms for 3SUM

3 Sum Closest

Problem

> Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers.

Analysis

思路，类似于 3 Sum 问题，不同之处在于寻找与target的绝对值最小的数；同样可以利用two pointers，对于 a + b + c 中的 b, c 来作为两个指针，a 为 num[i], 那么b初始值则为 num[i + 1], c初始值为num[len - 1]; 通过比较每次 a + b + c 的sum与target的大小，来确定移动b，或者移动c。

Solution

```
public class Solution {
    /**
    * @param numbers: Give an array numbers of n integer
    * @param target : An integer
    * @return : return the sum of the three integers, the sum cl
osest target.
    */
    public int threeSumClosest(int[] numbers,int target) {
        if (numbers == null || numbers.length < 3) {
            return Integer.MAX_VALUE;
        }

        Arrays.sort(numbers);

        int diff = Integer.MAX_VALUE / 2;
        int length = numbers.length;
        int sign = 1;

        for (int i = 0; i < length - 2; i++) {
```

```
            int pl = i + 1;
            int pr = length - 1;

            while (pl < pr) {
                int sum = numbers[i] + numbers[pl] + numbers[pr]
;
                if (sum == target) {
                    return sum;
                } else if (sum < target) {
                    if (target - sum < diff) {
                        diff = target - sum;
                        sign = -1;
                    }
                    pl++;
                } else {
                    if (sum - target < diff) {
                        diff = sum - target;
                        sign = 1;
                    }
                    pr--;
                }
            }
        }
        return target + sign * diff;
    }

    // Use Math.abs(), two pointers
    public int threeSumClosestV2(int[] numbers,int target) {
        if (numbers == null || numbers.length < 3) {
            return Integer.MAX_VALUE;
        }

        Arrays.sort(numbers);

        int length = numbers.length;
        int closest = Integer.MAX_VALUE / 2;

        for (int i = 0; i < length - 2; i++) {
            int pl = i + 1;
            int pr = length - 1;
```

```
            while (pl < pr) {
                int sum = numbers[i] + numbers[pl] + numbers[pr]
;
                if (sum == target) {
                    return sum;
                } else if (sum < target) {
                    pl++;
                } else {
                    pr--;
                }
                closest = Math.abs(sum - target) < Math.abs(clos
est - target) ?
                        sum : closest;
            }
        }
        return closest;
    }
}
```

Sort Colors II

LintCode Sort Colors II

Question

Given an array of n objects with k different colors (numbered from `1` to `k`), sort them so that objects of the same color are adjacent, with the colors in the order `1, 2, ... k`.

Analysis

简单的办法可以利用two pass, 相当于counting sort，计数每个color的个数，再依次填入到原来的array中；这种方法空间复杂度为 O(k), 时间复杂度为 O(n)。

第二种则是利用两个指针的方法，设定pl和pr，左右两个指针，初始位置分别为数组两端，pl = 0, pr = colors.length - 1. 同时，由于题目限制条件，已知min和max，因此可以据此作为比较，来决定如何移动pl，pr两个指针。不断对满足min和max条件的colors进行swap，就可以在in-place的条件下，做到sorting colors，这种算法的空间复杂度为O(1), 而时间复杂度：这种方法的时间复杂度为O(n^2): T(n) = T(n - 2) + n。

Solution

```
public class Solution {
    /**
     * Method I: O(k) space, O(n) time; two-pass algorithm, coun
ting sort
     * @param colors: A list of integer
     * @param k: An integer
     * @return: nothing
     */
    public void sortColors2TwoPass(int[] colors, int k) {
```

```
        int[] count = new int[k];
        for (int color : colors) {
            count[color-1]++;
        }
        int index = 0;
        for (int i = 0; i < k; i++) {
            while (count[i]>0) {
                colors[index++] = i+1;
                count[i]--;
            }
        }
    }

    /**
     * Method II:
     *  Each time sort the array into three parts:
     *  [all min] [all unsorted others] [all max],
     *  then update min and max and sort the [all unsorted other
s]
     *  with the same method.
     *
     * @param colors: A list of integer
     * @param k: An integer
     * @return: nothing
     */
    public void sortColors2(int[] colors, int k) {
        int pl = 0;
        int pr = colors.length - 1;
        int i = 0;
        int min = 1, max = k;
        while (min < max) {
            while (i <= pr) {
                if (colors[i] == min) {
                    swap(colors, pl, i);
                    i++;
                    pl++;
                } else if (colors[i] == max) {
                    swap(colors, pr, i);
                    pr--;
                } else {
```

```
                    i++;
                }
                // printArray(colors);
            }
            i = pl;
            min++;
            max--;
        }
    }

    private void swap(int[] colors, int i, int j) {
        int temp = colors[i];
        colors[i] = colors[j];
        colors[j] = temp;
    }

    public static void main(String[] args) {
        Solution s = new Solution();
        int[] colors = new int[]{2, 5, 3, 4, 2, 2, 1};
        int k = 5;
        s.sortColors2(colors, k);
    }
    private static void printArray(int[] a) {
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.print("\n");
    }
}
```

Reference:

http://blog.welkinlan.com/2015/08/25/sort-colors-i-ii-leetcode-lintcode-java/
http://www.cnblogs.com/yuzhangcmu/p/4177326.html
http://blog.csdn.net/nicaishibiantai/article/details/43306431

Majority Number

Question

> Given an array of integers, the majority number is the number that occurs more than half of the size of the array. Find it.
>
> Example
>
> Given [1, 1, 1, 1, 2, 2, 2], return 1
>
> Challenge
>
> O(n) time and O(1) extra space

Analysis

最直观的想法就是，建立HashMap，记录list中的每一个integer元素的个数，如果大于1/2 list长度，即可返回。

进一步分析发现，其实并不需要记录所有的integer元素的个数，可以只记录当前最多的那一个majority。这种方法也称为 wikipedia: Boyer–Moore majority vote algorithm

> The Boyer-Moore Vote Algorithm solves the majority vote problem in linear time O(n) and logarithmic space O(log n)

Moore's Page

> As we sweep we maintain a pair consisting of a current candidate and a counter. Initially, the current candidate is unknown and the counter is 0.
>
> When we move the pointer forward over an element e:
>
> - If the counter is 0, we set the current candidate to e and we set the counter to 1.
> - If the counter is not 0, we increment or decrement the counter according to whether e is the current candidate.
>
> When we are done, the current candidate is the majority element, if there is a majority.

换一种角度，是否可以直接从list中读取这个majority呢？如果对list进行排序，那么1/2处的元素，也就是majority的那个integer了。当然这种方法有个问题，就是对于没有majority的情况下（没有一个达到了1/2 总长度），是无法判断是否存在majority的，如果题目中明确一定存在这样的majority，那么这种方法也是可行的。

时间 O(n), 空间 O(1)

Solution

HashMap Solution

```
public class Solution {
    /**
     * @param nums: a list of integers
     * @return: find a majority number
     */
    public int majorityNumber(ArrayList<Integer> nums) {
        if (nums == null || nums.size() == 0) {
            return 0;
        }

        HashMap<Integer, Integer> map = new HashMap<Integer, Int
eger>();
        int listSize = nums.size();
        int halfSize = listSize / 2;
        int result = 0;

        for (int i = 0; i < listSize; i++) {
            int num = nums.get(i);

            if (!map.containsKey(num)) {
                map.put(num, 1);
            } else {
                map.put(num, 1 + map.get(num));
            }

            if (map.get(num) > halfSize) {
                result = num;
            }
        }

        return result;
    }
}
```

Boyer–Moore majority vote algorithm

```java
public class Solution {
    /**
     * @param nums: a array of integers
     * @return: find a majority element
     */
    public int majorityElement(int[] nums) {
        int n = nums.length;
        int candidate = nums[0], counter = 0;
        for (int i : nums) {
            if (counter == 0) {
                candidate = i;
                counter = 1;
            } else if (candidate == i) {
                counter++;
            } else {
                counter--;
            }
        }

        counter = 0;
        for (int i : nums) {
            if (i == candidate) counter++;
        }
        if (counter < (n + 1) / 2) return -1;
        return candidate;
    }
}
```

Majority Number II

Question

```
Given an array of integers, the majority number is the numbe
r that occurs more than 1/3 of the size of the array.

Find it.

Example
Given [1, 2, 1, 2, 1, 3, 3], return 1.

Note
There is only one majority number in the array.

Challenge
O(n) time and O(1) extra space.
```

Analysis

与Majority Number问题相似，最直观的想法就是，建立HashMap，记录list中的每一个integer元素的个数，如果大于1/3list长度，即可返回。

由于此时只要求大于1/3，Moore的方法需要稍加改变。满足大于[1/3]的数，应当小于等于2个，也就是说，需要maintain两个top number作为candidate。

时间 O(n)，空间 O(1)

Solution

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return: The majority number that occurs more than 1/3
     */
    public int majorityNumber(ArrayList<Integer> nums) {
        int count1 = 0, count2 = 0;
        int candidate1 = 0, candidate2 = 0;
        for (int num : nums) {
            if (num == candidate1) {
                count1++;
            } else if (num == candidate2) {
                count2++;
            } else if (count1 == 0) {
                candidate1 = num;
                count1 = 1;
            } else if (count2 == 0) {
                candidate2 = num;
                count2 = 1;
            } else {
                count1--;
                count2--;
            }
        }
        count1 = count2 = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums.get(i) == candidate1) {
                count1++;
            } else if (nums.get(i) == candidate2) {
                count2++;
            }
        }
        return count1 > count2 ? candidate1 : candidate2;
    }
}
```

Majority Number III

Question

> Given an array of integers and a number k, the majority number is the number that occurs more than 1/k of the size of the array.
>
> Example
>
> Given [3,1,2,3,2,3,3,4,4,4] and k=3, return 3.

Analysis

相比于Majority Number和Majority Number II问题，该问题更加一般化。最终所需的额外空间，也就与k有关。

Solution

```
public class Solution {
    /**
     * @param nums: A list of integers
     * @param k: As described
     * @return: The majority number
     */
    public int majorityNumber(ArrayList<Integer> nums, int k) {
        HashMap<Integer, Integer> counters = new HashMap<Integer
, Integer>();
        for (Integer num : nums) {
            if (!counters.containsKey(num)) {
                counters.put(num, 1);
            } else {
                counters.put(num, counters.get(num) + 1);
            }
            if (counters.size() >= k) {
```

```
                removeKey(counters);
            }
        }

        // corner case
        if (counters.size() == 0) {
            return Integer.MIN_VALUE;
        }

        // re-count the numbers
        for (Integer num : counters.keySet()) {
            counters.put(num, 0);
        }
        for (Integer num : nums) {
            if (counters.containsKey(num)) {
                counters.put(num, counters.get(num) + 1);
            }
        }

        // find majority
        int maxCounter = 0, maxKey = 0;
        for (Integer num : counters.keySet()) {
            if (counters.get(num) > maxCounter) {
                maxCounter = counters.get(num);
                maxKey = num;
            }
        }
        return maxKey;
    }

    private void removeKey(HashMap<Integer, Integer> counters) {
        Set<Integer> keySet = counters.keySet();
        List<Integer> removeList = new ArrayList<>();
        for (Integer key : keySet) {
            counters.put(key, counters.get(key) - 1);
            if (counters.get(key) == 0) {
                removeList.add(key);
            }
        }
        for (Integer key : removeList) {
```

```
                counters.remove(key);
            }
        }
    }
```

Best Time to Buy and Sell Stock

Question

http://www.lintcode.com/en/problem/best-time-to-buy-and-sell-stock/

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.
>
> Example
>
> Given an example [3,2,3,1,2], return 1

Analysis

最初的想法是问题可以转化一下：计算相邻两天的变化量array，再对这个差值array寻找maximum subarray，得到的maximum也就是对应的maximum profit

另一种想法是：仅记录loop过的历史最小值，再寻找当前profit最大值

Solution

```java
// Diff + Maximum Subarray

public class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int[] diff = new int[prices.length];

        diff[0] = 0;
        for (int i = 1; i < prices.length; i++) {
            diff[i] =  prices[i] - prices[i - 1];
        }
        return maxSubArray(diff);

    }

    public int maxSubArray(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        int length = nums.length;
        int max = Integer.MIN_VALUE;
        int sum = 0;
        for (int i = 0; i < length; i++) {
            sum = sum + nums[i];
            max = Math.max(sum, max);
            sum = Math.max(sum, 0);
        }
        return max;
    }
}
```

```
// Only remember the smallest price

public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }

        // Only remember the smallest price
        int min = Integer.MAX_VALUE;
        int profit = 0;
        for (int i : prices) {
            min = i < min ? i : min;
            profit = (i - min) > profit ? i - min : profit;
        }

        return profit;
    }
}
```

Reference

Best Time to Buy and Sell Stock Series: I, II, III, IV

来自梁佳宾的网络日志: Best Time to Buy and Sell Stock I II III IV

Best Time to Buy and Sell Stock II

Question

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
>
> Example
>
> Given an example [2,1,2,0,1], return 2

Analysis

很基本的转化问题，只需考虑所有相邻两天的增量是否为正值，如果为正值，则可以计入收益profit。

Solution

```java
class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }
        int profit = 0;
        for (int i = 0; i < prices.length - 1; i++) {
            int diff = prices[i + 1] - prices[i];
            if (diff > 0) {
                profit += diff;
            }
        }
        return profit;
    }
};
```

Best Time to Buy and Sell Stock III

Question

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/

> Say you have an array for which the ith element is the price of a given stock on day i.
>
> Design an algorithm to find the maximum profit. You may complete at most two transactions.
>
> Example
>
> Given an example [4,4,6,1,1,4,2,5], return 6.

Analysis

一种算法比较容易想到： 找寻一个点j，将原来的prices[0..n-1]分割为prices[0..j]和prices[j..n-1]，分别求两段的最大profit。 内层循环与Best Time to Buy and Sell Stock相同，为O(n), 外层循环为O(n)，记录不同的j两段maxProfit的值需要O(n)空间，因此总体时间复杂度为O(n^2)，空间复杂度为O(n)。

改进（参考pickless的讲解）：对于点j+1，求prices[0..j+1]的最大profit时，很多工作是重复的，在求prices[0..j]的maxProfit中已经计算过了。 类似于Best Time to Buy and Sell Stock，可以在O(1)的时间从prices[0..j]推出prices[0..j+1]的最大profit。 但是如何从prices[j..n-1]推出prices[j+1..n-1]？反过来思考，我们可以用O(1)的时间由prices[j+1..n-1]推出prices[j..n-1]。 最终算法： 数组l[i]记录了prices[0..i]的最大profit， 数组r[i]记录了prices[i..n]的最大profit。 已知l[i]，求l[i+1]是简单的，同样已知r[i]，求r[i-1]也很容易。 最后，我们再用O(n)的时间找出最大的l[i]+r[i]，即为题目所求。 这样时间复杂度O(n), 空间复杂度O(n)

参考 http://blog.csdn.net/fightforyourdream/article/details/14503469

Solution

A Intuitive Implementation Using Divide And Conquer

```
import java.util.*;

public class Solution {
    /**
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int[] prices) {
        // find maxProfit for {0, j}, find maxProfit for {j + 1,
 n - 1}
        // find max for {max{0, j}, max{j + 1, n - 1}}

        if (prices == null || prices.length == 0) {
            return 0;
        }

        int maximumProfit = 0;
        int n = prices.length;

        ArrayList<Profit> preMaxProfit = new ArrayList<Profit>(n
);
        ArrayList<Profit> postMaxProfit = new ArrayList<Profit>(
n);
        for (int i = 0; i < n; i++) {
            preMaxProfit.add(maxProfitHelper(prices, 0, i));
            postMaxProfit.add(maxProfitHelper(prices, i + 1, n -
1));
        }
        for (int i = 0; i < n; i++) {
            int profit = preMaxProfit.get(i).maxProfit + postMax
Profit.get(i).maxProfit;
            maximumProfit = Math.max(profit, maximumProfit);
        }
        return maximumProfit;
    }

    private Profit maxProfitHelper(int[] prices, int startIndex,
int endIndex) {
```

```
        int minPrice = Integer.MAX_VALUE;
        int maxProfit = 0;
        for (int i = startIndex; i <= endIndex; i++) {
            if (prices[i] < minPrice) {
                minPrice = prices[i];
            }
            if (prices[i] - minPrice > maxProfit) {
                maxProfit = prices[i] - minPrice;
            }
        }
        return new Profit(maxProfit, minPrice);
    }

    public static void main(String[] args) {
        int[] prices = new int[]{4,4,6,1,1,4,2,5};
        Solution s = new Solution();
        System.out.println(s.maxProfit(prices));
    }
};

class Profit {
    int maxProfit, minPrice;
    Profit(int maxProfit, int minPrice) {
        this.maxProfit = maxProfit;
        this.minPrice = minPrice;
    }
}
```

Another Implementation with Dynamic Programming

```
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length <= 1) {
            return 0;
        }

        int[] left = new int[prices.length];
        int[] right = new int[prices.length];

        // DP from left to right;
        left[0] = 0;
        int min = prices[0];
        for (int i = 1; i < prices.length; i++) {
            min = Math.min(prices[i], min);
            left[i] = Math.max(left[i - 1], prices[i] - min);
        }

        //DP from right to left;
        right[prices.length - 1] = 0;
        int max = prices[prices.length - 1];
        for (int i = prices.length - 2; i >= 0; i--) {
            max = Math.max(prices[i], max);
            right[i] = Math.max(right[i + 1], max - prices[i]);
        }

        int profit = 0;
        for (int i = 0; i < prices.length; i++){
            profit = Math.max(left[i] + right[i], profit);
        }

        return profit;
    }
}
```

Reference

http://liangjiabin.com/blog/2015/04/leetcode-best-time-to-buy-and-sell-stock.html
http://www.jiuzhang.com/solutions/best-time-to-buy-and-sell-stock-iii/

Best Time to Buy and Sell Stock IV

Question

> Say you have an array for which the `ith` element is the price of a given stock on day *i*.
>
> Design an algorithm to find the maximum profit. You may complete at most `k` transactions.
>
> Example
>
> Given prices = [4,4,6,1,1,4,2,5], and k = 2, return 6.

Analysis

> 下面的解法主要是能把两次的限制推广到k次交易：
>
> 这道题是Best Time to Buy and Sell Stock的扩展，现在我们最多可以进行两次交易。我们仍然使用动态规划来完成，事实上可以解决非常通用的情况，也就是最多进行k次交易的情况。 这里我们先解释最多可以进行k次交易的算法，然后最多进行两次我们只需要把k取成2即可。我们还是使用“局部最优和全局最优解法”。我们维护两种量，一个是当前到达第i天可以最多进行j次交易，最好的利润是多少（global[i][j]），另一个是当前到达第i天，最多可进行j次交易，并且最后一次交易在当天卖出的最好的利润是多少（local[i][j]）。下面我们来看递推式，全局的比较简单，
>
> global[i][j]=max(local[i][j],global[i-1][j])，
>
> 也就是去当前局部最好的，和过往全局最好的中大的那个（因为最后一次交易如果包含当前天一定在局部最好的里面，否则一定在过往全局最优的里面）。
>
> 全局（到达第i天进行j次交易的最大收益）= max{局部（在第i天交易后，恰好满足j次交易），全局（到达第i-1天时已经满足j次交易）}
>
> 对于局部变量的维护，递推式是
>
> local[i][j]=max(global[i-1][j-1]+max(diff,0),local[i-1][j]+diff)，

> 也就是看两个量，第一个是全局到i-1天进行j-1次交易，然后加上今天的交易，如果今天是赚钱的话（也就是前面只要j-1次交易，最后一次交易取当前天），第二个量则是取local第i-1天j次交易，然后加上今天的差值（这里因为local[i-1][j]比如包含第i-1天卖出的交易，所以现在变成第i天卖出，并不会增加交易次数，而且这里无论diff是不是大于0都一定要加上，因为否则就不满足local[i][j]必须在最后一天卖出的条件了）。
>
> 局部（在第i天交易后，总共交易了j次） = max{情况2，情况1}
>
> 情况1：在第i-1天时，恰好已经交易了j次（local[i-1][j]），那么如果i-1天到i天再交易一次：即在第i-1天买入，第i天卖出（diff），则这不并不会增加交易次数！【例如我在第一天买入，第二天卖出；然后第二天又买入，第三天再卖出的行为 和 第一天买入，第三天卖出 的效果是一样的，其实只进行了一次交易！因为有连续性】 情况2：第i-1天后，共交易了j-1次（global[i-1][j-1]），因此为了满足“第i天过后共进行了j次交易，且第i天必须进行交易”的条件：我们可以选择1：在第i-1天买入，然后再第i天卖出（diff），或者选择在第i天买入，然后同样在第i天卖出（收益为0）。
>
> 上面的算法中对于天数需要一次扫描，而每次要对交易次数进行递推式求解，所以时间复杂度是O(n*k)，如果是最多进行两次交易，那么复杂度还是O(n)。空间上只需要维护当天数据皆可以，所以是O(k)，当k=2，则是O(1)。
>
> 补充：这道题还有一个陷阱，就是当k大于天数时，其实就退化成 Best Time to Buy and Sell Stock II 了。就不能用动规来做了，为什么？（请思考） 另外，Best Time to Buy and Sell Stock III 就是本题k=2的情况，所以说IV是II和III的综合。

http://blog.csdn.net/fightforyourdream/article/details/14503469
http://blog.csdn.net/linhuanmars/article/details/23236995
http://liangjiabin.com/blog/2015/04/leetcode-best-time-to-buy-and-sell-stock.html

Solution

Dynamic Programming 2D array implementation

```
class Solution {
    /**
     * @param k: An integer
```

```
     * @param prices: Given an integer array
     * @return: Maximum profit
     */
    public int maxProfit(int k, int[] prices) {
        if (prices == null || prices.length < 2) {
            return 0;
        }
        int days = prices.length;

        if (days <= k) {
            return maxProfit2(prices);
        }
        // local[i][j] 表示前i天，至多进行j次交易，第i天必须sell的最大
获益
        int[][] local = new int[days][k + 1];
        // global[i][j] 表示前i天，至多进行j次交易，第i天可以不sell的最
大获益
        int[][] global = new int[days][k + 1];

        for (int i = 1; i < days; i++) {
            int diff = prices[i] - prices[i - 1];
            for (int j = 1; j <= k; j++) {
                local[i][j] = Math.max(global[i - 1][j-1] + diff
,
                        local[i - 1][j] + diff);
                global[i][j] = Math.max(global[i - 1][j], local[
i][j]);
            }
        }
        return global[days - 1][k];
    }

    public int maxProfit2(int[] prices) {
        int maxProfit = 0;
        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i-1]) {
                maxProfit += prices[i] - prices[i-1];
            }
        }
        return maxProfit;
```

```
    }
};
```

Dynamic Programming 1D array implementation

```
public class Solution {
    public int maxProfit(int k, int[] prices) {
        if (prices.length < 2) return 0;
        if (k >= prices.length) return maxProfit2(prices);

        int[] local = new int[k + 1];
        int[] global = new int[k + 1];

        for (int i = 1; i < prices.length ; i++) {
            int diff = prices[i] - prices[i - 1];

            for (int j = k; j > 0; j--) {
                local[j] = Math.max(global[j - 1], local[j] + di
ff);
                global[j] = Math.max(global[j], local[j]);
            }
        }

        return global[k];
    }


    public int maxProfit2(int[] prices) {
        int maxProfit = 0;

        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1]) {
                maxProfit += prices[i] - prices[i - 1];
            }
        }

        return maxProfit;
    }
}
```

Data Structure

HashMap

HashMap 的两种遍历方式 第一种

```
Map map = new HashMap();
Iterator iter = map.entrySet().iterator();
while (iter.hasNext()) {
Map.Entry entry = (Map.Entry) iter.next();
Object key = entry.getKey();
Object val = entry.getValue();
}
```

效率高,以后一定要使用此种方式！

第二种

```
Map map = new HashMap();
Iterator iter = map.keySet().iterator();
while (iter.hasNext()) {
Object key = iter.next();
Object val = map.get(key);
}
```

效率低,以后尽量少使用！

Union Find (Disjoint Set)

并查集: 一种用来解决集合查询合并的数据结构 支持O(1) find / O(1) union

并查集可以干什么?

1. 判断在不在同一个集合中。

    - find 操作
2. 关于集合合并
    - union 操作

并查集的操作

1. 查询 Find (递归? 非递归?)

模板代码

```
HashMap<Integer, Integer> father = new HashMap<Integer, Integer>
();

int find(int x) {
    int parent = x;
    while (parent ! = father.get(parent)) {
        parent = father.get(parent);
    }
    return parent;
}
```

1. 合并 Union

老大哥之间合并 跟小弟没关系

```
HashMap<Integer, Integer> father = new HashMap<Integer. Integer>
();

void union(int x, int y) {
    int fa_x = find(x);
    int fa_y = find(y);
    if (fa_x != fa_y) {
        father.put(fa_x, fa_y);
    }
}
```

并查集完整模板

```
class UnionFind {
    UnionFind() {}
    HashMap<Integer, Integer> father = new HashMap<Integer. Inte
ger>();
    int find(int x) {
        int parent = x;
        while (parent ! = father.get(parent)) {
            parent = father.get(parent);
        }
        return parent;
    }
    void union(int x, int y) {
        int fa_x = find(x);
        int fa_y = find(y);
        if (fa_x != fa_y) {
            father.put(fa_x, fa_y);
        }
    }
}
```

Trie

Linked List

`LinkedList` and `ArrayList` are two different implementations of the List interface. `LinkedList` implements it with a doubly-linked list. `ArrayList` implements it with a dynamically re-sizing array.

`LinkedList<E>` allows for constant-time insertions or removals using iterators, but only sequential access of elements.

`ArrayList<E>` , on the other hand, allow fast random read access, so you can grab any element in constant time.

- When to use LinkedList over ArrayList?

Heap

Heap

A min-heap is a binary tree such that

- the data contained in each node is less than (or equal to) the data in that node's children.
- the binary tree is complete

A max-heap is a binary tree such that

- the data contained in each node is greater than (or equal to) the data in that node's children.
- the binary tree is complete

Sift Up

```
void siftup(int id) {
    while (parent(id) > -1) {
        int parentId = parent(id);
        if (comparesmall(heap.get(parentId), heap.get(id)) == tr
ue) {
            break;
        } else {
            swap(id, parentId);
        }
        id = parentId;
    }
}
```

Sift Down

```
void siftdown(int id) {
    while (lson(id) < heap.size()) {
        int leftId = lson(id);
        int rightId = rson(id);
        int son;
        if (rightId >= heap.size() || (comparesmall(heap.get(lef
tId), heap.get(rightId)) == true)) {
            son = leftId;
        } else {
            son = rightId;
        }

        if (comparesmall(heap.get(id), heap.get(son)) == true) {
            break;
        } else {
            swap(id, son);
        }
        id = son;
    }
}
```

Resources

- bubkoo.com 常见排序算法 - 堆排序 (Heap Sort)
- Data Structures Heap, Heap Sort & Priority Queue
- Priority Queue Implementation
- CMU: Trees Heaps & Other Trees

Stack

Queue

Hash Function

Question

> In data structure Hash, hash function is used to convert a string(or any other type) into an integer smaller than hash size and bigger or equal to zero. The objective of designing a hash function is to "hash" the key as unreasonable as possible. A good hash function can avoid collision as less as possible. A widely used hash function algorithm is using a magic number `33`, consider any string as a `33` based big integer like follow:

```
hashcode("abcd") = (ascii(a) * 333 + ascii(b) * 332 + ascii(c) *
33 + ascii(d)) % HASH_SIZE

                                 = (97* 333 + 98 * 332 + 99 * 33 +1
00) % HASH_SIZE

                                 = 3595978 % HASH_SIZE
```

> here HASH_SIZE is the capacity of the hash table (you can assume a hash table is like an array with index 0 ~ HASH_SIZE-1).
>
> Given a string as a key and the size of hash table, return the hash value of this key.

Clarification

For this problem, you are not necessary to design your own hash algorithm or consider any collision issue, you just need to implement the algorithm as described.

Example

For key="abcd" and size=100, return 78

Analysis

直白的实现方法就是按照定义对sum进行HASH_SIZE取模运算，但是实际上这样会产生溢出。于是应用模运算的法则，每一次累加sum时，就可以取模HASH_SIZE，这样就可以很显著地减小最终的sum值。

需要注意到是定义sum为long类型，最终返回时转化为int。

参考： http://baike.baidu.com/view/2385246.htm
https://en.wikipedia.org/wiki/Modular_arithmetic

Solutions

```
class Solution {
    /**
     * @param key: A String you should hash
     * @param HASH_SIZE: An integer
     * @return an integer
     */
    public int hashCode(char[] key,int HASH_SIZE) {
        int N = key.length;
        long sum = 0;
        for (int i = 0; i < N; i++) {
            sum = (sum * 33 + (int) (key[i])) % HASH_SIZE;
        }

        return (int) (sum);
    }
};
```

Heapify

Question

Given an integer array, heapify it into a min-heap array.

For a heap array A, A[0] is the root of heap, and for each A[i], A[i *2 + 1] is the left child of A[i] and A[i* 2 + 2] is the right child of A[i].

Clarification

What is heap?

- Heap is a data structure, which usually have three methods: push, pop and top. where "push" add a new element the heap, "pop" delete the minimum/maximum element in the heap, "top" return the minimum/maximum element.

What is heapify?

- Convert an unordered integer array into a heap array. If it is min-heap, for each element A[i], we will get A[i *2 + 1] >= A[i] and A[i* 2 + 2] >= A[i].

What if there is a lot of solutions?

- Return any of them.

Example

Given [3,2,1,4,5], return [1,2,3,4,5] or any legal heap array.

Challenge

O(n) time complexity

Analysis

Heapify一个Array，也就是对array中的元素进行siftup或者siftdown的操作。根据min heap定义进行操作即可。

这里值得注意的是，对于扫描整个array的情况下，siftup和siftdown有complexity上的区别。

基本的原因在于：siftdown的complexity，实质上是node相对于bottom移动的次数，而根据binary heap本身的特性，决定了约靠近bottom的node越多；相对照的是siftup，是node相对于root节点的移动次数。

> The number of operations required for each operation is proportional to the distance the node may have to move. For siftDown, it is the distance from the bottom of the tree, so siftDown is expensive for nodes at the top of the tree. With siftUp, the work is proportional to the distance from the top of the tree, so siftUp is expensive for nodes at the bottom of the tree. Although both operations are O(log n) in the worst case, in a heap, only one node is at the top whereas half the nodes lie in the bottom layer. **So it shouldn't be too surprising that if we have to apply an operation to every node, we would prefer siftDown over siftUp**

关于其中具体的理由以及分析方法，可以参考stackoverflow上一个问答：
http://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity

如果Heapify可以用O(n)实现，那么HeapSort所需的时间复杂度为何是O(nlogn)？因为HeapSort其实包含了两个步骤，第一步，Heapify，build (min) heap，所需时间复杂度O(n)，第二步，依次删除最小值（min heap），对于Heap来说，删除操作的复杂度是O(logn)， 而这个删除需要执行O(n)，来得到最终sort的结果，于是总体时间复杂度是O(nlogn)。

Solution

Siftup O(nlogn)

```
public class Solution {
    /**
     * @param A: Given an integer array
     * @return: void
     */

    private void swap(int[] A, int i, int j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }

    private void siftup(int[] A, int k) {
        while (k != 0) {
            int parent = (k - 1) / 2;
            if (A[k] > A[parent]) {
                break;
            }
            swap(A, k, parent);
            k = parent;
        }
    }

    public void heapify(int[] A) {
        for (int i = 0; i < A.length; i++) {
            siftup(A, i);
        }
    }
}
```

Siftdown O(n)

```
public class Solution {
    /**
     * @param A: Given an integer array
     * @return: void
     */
    private void siftdown(int[] A, int k) {
        while (k < A.length) {
            int smallest = k;
            if (k * 2 + 1 < A.length && A[k * 2 + 1] < A[smalles
t]) {
                smallest = k * 2 + 1;
            }
            if (k * 2 + 2 < A.length && A[k * 2 + 2] < A[smalles
t]) {
                smallest = k * 2 + 2;
            }
            if (smallest == k) {
                break;
            }
            int temp = A[smallest];
            A[smallest] = A[k];
            A[k] = temp;

            k = smallest;
        }
    }

    public void heapify(int[] A) {
        for (int i = A.length / 2; i >= 0; i--) {
            siftdown(A, i);
        }
    }
}
```

Reference

- Heapify demo
- HeapSort Analysis

LRU Cache

Question

> Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: `get` and `set`.
>
> `get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
>
> `set(key, value)` - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Analysis

LRU，也就是least recently used，最近使用最少的；这样一个数据结构，能够保持一定的顺序，使得最近使用过的时间或者顺序被记录，实际上，具体每一个item最近一次何时被使用的，并不重要，重要的是在这样的一个结构中，item的相对位置代表了最近使用的顺序；满足这样考虑的结构可以是链表list或者数组array，不过前者更有利于insert和delete的操纵，此外，需要记录这个链表的head和tail，方便进行移动到tail或者删除head的操作，即：head.next作为最近最少使用的item，tail.prev为最近使用过的item，在set时，如果超出capacity，则删除head.next，同时将要插入的item放入tail.prev, 而get时，如果存在，只需把item更新到tail.prev即可。

这样set与get均为O(1)时间的操作 （HashMap Get/Set + LinkedList Insert/Delete)，空间复杂度为O(n), n为capacity。

Solution

```
public class LRUCache {
    private class Node {
```

```
        Node prev;
        Node next;
        int key;
        int value;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
            this.prev = null;
            this.next = null;
        }
    }

    private int capacity;
    private HashMap<Integer, Node> hm = new HashMap<Integer, Nod
e>();
    private Node head = new Node(-1, -1);
    private Node tail = new Node(-1, -1);

    // @param capacity, an integer
    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.head.next = this.tail;
        this.tail.prev = this.head;
    }

    // @return an integer
    public int get(int key) {
        if (!hm.containsKey(key)) {
            return -1;
        }
        Node current = hm.get(key);
        current.prev.next = current.next;
        current.next.prev = current.prev;

        moveToTail(current);

        return hm.get(key).value;
    }
```

```
    // @param key, an integer
    // @param value, an integer
    // @return nothing
    public void set(int key, int value) {
        if (get(key) != -1) {
            hm.get(key).value = value;
            return;
        }
        if (hm.size() == capacity) {
            hm.remove(head.next.key);
            head.next = head.next.next;
            head.next.prev = head;
        }
        Node insert = new Node(key, value);
        hm.put(key, insert);
        moveToTail(insert);
    }

    private void moveToTail(Node current) {
        current.next = tail;
        tail.prev.next = current;
        current.prev = tail.prev;
        tail.prev = current;
    }
}
```

LFU Cache

Question

LFU (Least Frequently Used) is a famous cache eviction algorithm.

For a cache with capacity k, if the cache is full and need to evict a key in it, the key with the lease frequently used will be kicked out.

Implement set and get method for LFU cache.

Have you met this question in a real interview? Yes

Example

Given capacity=3

```
set(2,2)
set(1,1)
get(2)
>> 2
get(1)
>> 1
get(2)
>> 2
set(3,3)
set(4,4)
get(3)
>> -1
get(2)
>> 2
get(1)
>> 1
get(4)
>> 4
```

Analysis

Solution

```
public class LFUCache {

    private final Map<Integer, CacheNode> cache;
    private final LinkedHashSet[] frequencyList;
    private int lowestFrequency;
    private int maxFrequency;
    private final int maxCacheSize;

    // @param capacity, an integer
    public LFUCache(int capacity) {
        // Write your code here
        this.cache = new HashMap<Integer, CacheNode>(capacity);
        this.frequencyList = new LinkedHashSet[capacity * 2];
        this.lowestFrequency = 0;
        this.maxFrequency = capacity * 2 - 1;
        this.maxCacheSize = capacity;
        initFrequencyList();
    }

    // @param key, an integer
    // @param value, an integer
    // @return nothing
    public void set(int key, int value) {
        // Write your code here
        CacheNode currentNode = cache.get(key);
        if (currentNode == null) {
            if (cache.size() == maxCacheSize) {
                doEviction();
            }
            LinkedHashSet<CacheNode> nodes = frequencyList[0];
            currentNode = new CacheNode(key, value, 0);
            nodes.add(currentNode);
            cache.put(key, currentNode);
            lowestFrequency = 0;
        } else {
            currentNode.v = value;
```

```
        }
        addFrequency(currentNode);
    }

    public int get(int key) {
        // Write your code here
        CacheNode currentNode = cache.get(key);
        if (currentNode != null) {
            addFrequency(currentNode);
            return currentNode.v;
        } else {
            return -1;
        }
    }

    public void addFrequency(CacheNode currentNode) {
        int currentFrequency = currentNode.frequency;
        if (currentFrequency < maxFrequency) {
            int nextFrequency = currentFrequency + 1;
            LinkedHashSet<CacheNode> currentNodes = frequencyLis
t[currentFrequency];
            LinkedHashSet<CacheNode> newNodes = frequencyList[ne
xtFrequency];
            moveToNextFrequency(currentNode, nextFrequency, curr
entNodes, newNodes);
            cache.put(currentNode.k, currentNode);
            if (lowestFrequency == currentFrequency && currentNo
des.isEmpty()) {
                lowestFrequency = nextFrequency;
            }
        } else {
            // Hybrid with LRU: put most recently accessed ahead
 of others:
            LinkedHashSet<CacheNode> nodes = frequencyList[curre
ntFrequency];
            nodes.remove(currentNode);
            nodes.add(currentNode);
        }
    }
```

```
    public int remove(int key) {
        CacheNode currentNode = cache.remove(key);
        if (currentNode != null) {
            LinkedHashSet<CacheNode> nodes = frequencyList[curre
ntNode.frequency];
            nodes.remove(currentNode);
            if (lowestFrequency == currentNode.frequency) {
                findNextLowestFrequency();
            }
            return currentNode.v;
        } else {
            return -1;
        }
    }

    public int frequencyOf(int key) {
        CacheNode node = cache.get(key);
        if (node != null) {
            return node.frequency + 1;
        } else {
            return 0;
        }
    }

    public void clear() {
        for (int i = 0; i <= maxFrequency; i++) {
            frequencyList[i].clear();
        }
        cache.clear();
        lowestFrequency = 0;
    }

    public int size() {
        return cache.size();
    }

    public boolean isEmpty() {
        return this.cache.isEmpty();
    }
```

```
    public boolean containsKey(int key) {
        return this.cache.containsKey(key);
    }

    private void initFrequencyList() {
        for (int i = 0; i <= maxFrequency; i++) {
            frequencyList[i] = new LinkedHashSet<CacheNode>();
        }
    }

    private void doEviction() {
        int currentlyDeleted = 0;
        double target = 1; // just one
        while (currentlyDeleted < target) {
            LinkedHashSet<CacheNode> nodes = frequencyList[lowes
tFrequency];
            if (nodes.isEmpty()) {
                continue;
            } else {
                Iterator<CacheNode> it = nodes.iterator();
                while (it.hasNext() && currentlyDeleted++ < targ
et) {
                    CacheNode node = it.next();
                    it.remove();
                    cache.remove(node.k);
                }
                if (!it.hasNext()) {
                    findNextLowestFrequency();
                }
            }
        }
    }

    private void moveToNextFrequency(CacheNode currentNode, int
nextFrequency,
                                     LinkedHashSet<CacheNode> cu
rrentNodes,
                                     LinkedHashSet<CacheNode> ne
wNodes) {
        currentNodes.remove(currentNode);
```

```
            newNodes.add(currentNode);
            currentNode.frequency = nextFrequency;
        }

        private void findNextLowestFrequency() {
            while (lowestFrequency <= maxFrequency && frequencyList[
lowestFrequency].isEmpty()) {
                lowestFrequency++;
            }
            if (lowestFrequency > maxFrequency) {
                lowestFrequency = 0;
            }
        }

        private class CacheNode {
            public final int k;
            public int v;
            public int frequency;

            public CacheNode(int k, int v, int frequency) {
                this.k = k;
                this.v = v;
                this.frequency = frequency;
            }

        }
}
```

Reference

- An O(1) algorithm for implementing the LFU cache eviction scheme
- Java Articles: LFU Cache
- activemq LFUCache.java

Top k Largest Numbers

Question

> Given an integer array, find the top k largest numbers in it.

Example

Given [3,10,1000,-99,4,100] and k = 3. Return [1000, 100, 10].

Analysis

此题运用Priority Queue，也就是Max Heap来求解十分简洁。重点在于对该数据结构的理解和应用。 使用Max Heap，也就是保证了堆顶的元素是整个堆最大的那一个。第一步，先构建一个capacity为k的Max Heap，这需要O(n)时间；第二步，对这个Max Heap进行k次poll()操作，从中取出k个maximum的元素，这一步操作需要O(k *logn)*。因此最终整个操作的时间复杂度是 *O(n + k* logn)，空间复杂度是O(k)

对于Top k largest(or smallest) elements in an array问题，有如下几种常见方法：

1. 冒泡排序k次
2. 使用临时数组
3. 使用排序
4. 使用最大堆
5. 使用排序统计
6. 使用最小堆

(摘自GeeksForGeeks)

Method 1. Use bubble k times

Thanks to Shailendra for suggesting this approach.

1) Modify Bubble Sort to run the outer loop at most k times.

2) Print the last k elements of the array obtained in step 1.

Time Complexity: O(nk)

Like Bubble sort, other sorting algorithms like Selection Sort can also be modified to get the k largest elements.

Method 2. Use temporary array

K largest elements from arr[0..n-1]

1) Store the first k elements in a temporary array temp[0..k-1].

2) Find the smallest element in temp[], let the smallest element be min.

3) For each element x in arr[k] to arr[n-1] If x is greater than the min then remove min from temp[] and insert x.

4) Print final k elements of temp[]

Time Complexity: O((n-k) *k). If we want the output sorted then O((n-k)* k + klogk)

Method 3. Use sorting

1) Sort the elements in descending order in O(nLogn)

2) Print the first k numbers of the sorted array O(k).

Time complexity: O(nlogn)

Method 4. Use Max heap

1) Build a Max Heap tree in O(n)

2) Use Extract Max k times to get k maximum elements from the Max Heap O(klogn)

Time complexity: O(n + klogn)

Method 5. Use Order Statistics

1) Use order statistic algorithm to find the kth largest element. Please see the topic selection in worst-case linear time O(n) 2) Use QuickSort Partition algorithm to partition around the kth largest number O(n). 3) Sort the k-1 elements (elements greater than the kth largest element) O(kLogk). This step is needed only if sorted output is required.

Time complexity: O(n) if we don’t need the sorted output, otherwise O(n+kLogk)

Thanks to Shilpi for suggesting the first two approaches.

Method 6. Use Min Heap

This method is mainly an optimization of method 2 temp array. Instead of using temp[] array, use Min Heap.

Thanks to geek4u for suggesting this method.

1) Build a Min Heap MH of the first k elements (arr[0] to arr[k-1]) of the given array. O(k)

2) For each element, after the kth element (arr[k] to arr[n-1]), compare it with root of MH. ……a) If the element is greater than the root then make it root and call heapify for MH ……b) Else ignore it.

The step 2 is O((n-k) * logk)

3) Finally, MH has k largest elements and root of the MH is the kth largest element.

Time Complexity: O(k + (n-k)Logk) without sorted output. If sorted output is needed then O(k + (n-k)Logk + kLogk)

All of the above methods can also be used to find the kth largest (or smallest) element.

Solution

Max Heap

```
class Solution {
    /*
     * @param nums an integer array
     * @param k an integer
     * @return the top k largest numbers in array
     */
     public int[] topk(int[] nums, int k) {
         Comparator<Integer> comparator = new Comparator<Integer
>() {
             @Override
             public int compare(Integer o1, Integer o2) {
                 if(o1 < o2) {
                     return 1;
                 } else if(o1 > o2) {
                     return -1;
                 } else {
                     return 0;
                 }
             }
         };
         PriorityQueue<Integer> minheap = new PriorityQueue<Inte
ger>(k, comparator);

         for (int i : nums) {
             minheap.add(i);
         }

         int[] result = new int[k];
         for (int i = 0; i < result.length; i++) {
             result[i] = minheap.poll();
         }
         return result;
    }
};
```

Min Heap

```
class Solution {
    /*
     * @param nums an integer array
     * @param k an integer
     * @return the top k largest numbers in array
     */
    public int[] topk(int[] nums, int k) {
        PriorityQueue<Integer> minheap = new PriorityQueue<Integ
er>();
        for (int i = 0; i < k; i++) {
           minheap.offer(nums[i]);
        }
        for (int i = k; i < nums.length; i++) {
            if (nums[i] > minheap.peek()) {
                minheap.poll();
                minheap.offer(nums[i]);
            }
        }
        Iterator it = minheap.iterator();
        List<Integer> result = new ArrayList<Integer>();
        while(it.hasNext()) {
            result.add((Integer) it.next());
        }
        Collections.sort(result, Collections.reverseOrder());

        int[] res = new int[result.size()];
        Iterator<Integer> iter = result.iterator();
        for (int i = 0; i < res.length; i++) {
            res[i] = iter.next().intValue();
        }
        return res;
    }
};
```

Sorting

```
class Solution {
    /*
     * @param nums an integer array
     * @param k an integer
     * @return the top k largest numbers in array
     */
    public int[] topk(int[] nums, int k) {
        Arrays.sort(nums);
        int[] result = new int[k];
        int j = 0;
        for (int i = nums.length - 1; i > nums.length - k - 1; i
--) {
            result[j] = nums[i];
            j++;
        }
        return result;
    }
};
```

Reference

- 基于堆实现的优先级队列：PriorityQueue 解决 Top K 问题
- geeksforgeeks.org: k largest(or smallest) elements in an array
- My Favorite Interview Question by Arden Dertat - "In an integer array with N elements (N is large), find the minimum k elements (k << N)."

Top k Largest Numbers II

Question

> Implement a data structure, provide two interfaces:
>
> 1. `add(number)`. Add a new number in the data structure.
> 2. `topk()`. Return the top `k` largest numbers in this data structure. `k` is given when we create the data structure.
>
> Example

>

```
s = new Solution(3);
>> create a new data structure.
s.add(3)
s.add(10)
s.topk()
>> return [10, 3]
s.add(1000)
s.add(-99)
s.topk()
>> return [1000, 10, 3]
s.add(4)
s.topk()
>> return [1000, 10, 4]
s.add(100)
s.topk()
>> return [1000, 100, 10]
```

Analysis

与top k问题1类似，不太一样的一点在于动态添加，使用min heap来实现，能够比较好地通过更新min heap来记录top k。

当添加的元素数目在1 ~ k时，直接插入这一元素到min heap中； 当添加的元素数目超出k时，对于新添加的元素，需要与min heap的根进行比较，如果比minheap.peek()大，那么便删除根，添加该新元素。

Solution

```
public class Solution {
    private PriorityQueue<Integer> minheap;
    private int maxSize;

    public Solution(int k) {
        minheap = new PriorityQueue<Integer>();
        maxSize = k;
    }

    public void add(int num) {
        if (minheap.size() < maxSize) {
            minheap.offer(num);
        } else {
            if (num > minheap.peek()) {
                minheap.poll();
                minheap.offer(num);
            }
        }

    }

    public List<Integer> topk() {
        Iterator iter = minheap.iterator();
        List<Integer> result = new ArrayList<Integer>();
        while (iter.hasNext()) {
            result.add((Integer) iter.next());
        }
        Collections.sort(result, Collections.reverseOrder());
        return result;
    }
};
```

Kth Smallest Number in Sorted Matrix

Find the kth smallest number in a row and column sorted matrix.

Example

Given k = 4 and a matrix:

```
[
  [1 ,5 ,7],
  [3 ,7 ,8],
  [4 ,8 ,9],
]
```

return `5`

Challenge

O(k log n), n is the maximal number in width and height.

Tags

`Heap` `Priority Queue` `Matrix`

Related Problems

Hard Kth Smallest Sum In Two Sorted Arrays

Medium Kth Largest Element

Analysis

寻找第k小的数，可以联想到转化为数组后排序，不过这样的时间复杂度较高：O(n^2 log n^2) + O(k).

进一步，换种思路，考虑到堆（Heap）的特性，可以建立一个Min Heap，然后poll k次，得到第k个最小数字。不过这样的复杂度仍然较高。

考虑到问题中矩阵本身的特点：排过序，那么可以进一步优化算法。

```
[1 ,5 ,7],
[3 ,7 ,8],
[4 ,8 ,9],
```

因为行row和列column都已排序，那么matrix中最小的数字无疑是左上角的那一个，坐标表示也就是(0, 0)。寻找第2小的数字，也就需要在(0, 1), (1, 0)中得出；以此类推第3小的数字，也就要在(0, 1), (1, 0), (2, 0), (1, 1), (0, 2)中寻找。

在一个数字集合中寻找最大（Max）或者最小值（Min），很快可以联想到用Heap，在Java中的实现是Priority Queue，它的pop，push操作均为O(logn)，而top操作，得到堆顶仅需O(1)。

从左上(0, 0)位置开始往右/下方遍历，使用一个Hashmap记录visit过的坐标，把候选的数字以及其坐标放入一个大小为k的heap中（只把未曾visit过的坐标放入heap），并且每次放入前弹出掉（poll）堆顶元素，这样最多会添加（push）2k个元素。时间复杂度是O(klog2k)，也就是说在矩阵自身特征的条件上优化，可以达到常数时间的复杂度，空间复杂度也为O(k)，即存储k个候选数字的Priority Queue（Heap）。

Solution

```
class Number {
    public int x, y, val;
    public Number(int x, int y, int val) {
        this.x = x;
        this.y = y;
        this.val = val;
    }
}

class NumberComparator implements Comparator<Number> {
    public int compare(Number a, Number b) {
        return a.val - b.val;
    }
}

public class Solution {
```

```
    private boolean isValid(int x, int y, int[][] matrix, boolean
[][] visited) {
        if (x < matrix.length && y < matrix[x].length && !visite
d[x][y]) {
            return true;
        }
        return false;
    }

    int[] dx = new int[] {0, 1};
    int[] dy = new int[] {1, 0};

    /**
     * @param matrix: a matrix of integers
     * @param k: an integer
     * @return: the kth smallest number in the matrix
     */
    public int kthSmallest(int[][] matrix, int k) {
        // Validate input
        if (matrix == null || matrix.length == 0) {
            return -1;
        }
        if (matrix.length * matrix[0].length < k) {
            return -1;
        }

        // Define min heap
        PriorityQueue<Number> heap = new PriorityQueue<Number>(k
, new NumberComparator());

        heap.add(new Number(0, 0, matrix[0][0]));

        // Define visited matrix
        boolean[][] visited = new boolean[matrix.length][matrix[0
].length];

        visited[0][0] = true;

        for (int i = 0; i < k - 1; i++) {
            Number smallest = heap.poll();
```

```
            for (int j = 0; j < 2; j++) {
                // Next coordinates
                int nx = smallest.x + dx[j];
                int ny = smallest.y + dy[j];

                if (isValid(nx, ny, matrix, visited)) {
                    visited[nx][ny] = true;
                    heap.add(new Number(nx, ny, matrix[nx][ny]))
;
                }
            }
        }

        return heap.peek().val;
    }
}
```

Reference

- Kth Smallest Number in Sorted Matrix
- Jiuzhang

Kth Smallest Sum In Two Sorted Arrays

Difficulty Hard Accepted Rate 19%

Question

Given two integer arrays sorted in ascending order and an integer k. Define sum = a + b, where a is an element from the first array and b is an element from the second one. Find the kth smallest sum out of all possible sums.

Example

Given [1, 7, 11] and [2, 4, 6].

For k = 3, return 7.

For k = 4, return 9.

For k = 8, return 15.

Challenge Do it in either of the following time complexity:

O(k log min(n, m, k)). where n is the size of A, and m is the size of B. O((m + n) log maxValue). where maxValue is the max number in A and B.

Tags

Heap Priority Queue Sorted Matrix

Related Problems

Medium Kth Smallest Number in Sorted Matrix Medium Search a 2D Matrix II

Analysis

此题乍看似乎没有很好的思路，其实稍加转化就变成了熟悉的问题：Kth Smallest Number in Sorted Matrix.

两个array中间分别取一个数相加得到一个sum，其实可以想象一个Matrix，里面元素的坐标就是分别在两个Array中的下标index，而元素的值则是sum的值。那么很显然，因为两个array都是排序过的，那么对于这个想象中的matrix中的每一行每一列来说，都是排好序的。

比如对于 `[1, 7, 11]` and `[2, 4, 6]` .

```
M   1,  7, 11
2,  3,  9, 13
4,  5, 11, 15
6,  7, 13, 17
```

接下来就可以利用PriorityQueue构造Min Heap来解决了。

Solution

```
class Node {
    public int x, y, sum;
    public Node(int x, int y, int sum) {
        this.x = x;
        this.y = y;
        this.sum = sum;
    }
}

class NodeComparator implements Comparator<Node> {
    @Override
    public int compare(Node a, Node b) {
        return a.sum - b.sum;
    }
}

public class Solution {
    int[] dx = new int[] {0, 1};
    int[] dy = new int[] {1, 0};

    // Check if a coordinate is valid and should be marked as vi
sited
```

```
    public boolean isValid(int x, int y, int[] A, int[] B, boole
an[][] visited) {
        if (x < A.length && y < B.length && !visited[x][y]) {
            return true;
        }
        return false;
    }

    /**
     * @param A an integer arrays sorted in ascending order
     * @param B an integer arrays sorted in ascending order
     * @param k an integer
     * @return an integer
     */
    public int kthSmallestSum(int[] A, int[] B, int k) {
        // Validation of input
        if (A == null || B == null || A.length == 0 || B.length
== 0) {
            return -1;
        }
        if (A.length * B.length < k) {
            return -1;
        }

        PriorityQueue<Node> heap = new PriorityQueue<Node>(k, new
 NodeComparator());
        heap.offer(new Node(0, 0, A[0] + B[0]));

        boolean[][] visited = new boolean[A.length][B.length];

        for (int i = 0; i < k - 1; i++) {
            Node smallest = heap.poll();
            for (int j = 0; j < 2; j++) {
                int nextX = smallest.x + dx[j];
                int nextY = smallest.y + dy[j];

                if (isValid(nextX, nextY, A, B, visited)) {
                    visited[nextX][nextY] = true;
                    int nextSum = A[nextX] + B[nextY];
                    heap.offer(new Node(nextX, nextY, nextSum));
```

```
                    }
                }
            }
            return heap.peek().sum;
        }
    }
```

Reference

Kth Largest Element

Question

Find K-th largest element in an array.

Notice

You can swap elements in the array

Example

In array [9,3,2,4,8], the 3rd largest element is 4.

In array [1,2,3,4,5], the 1st largest element is 5, 2nd largest element is 4, 3rd largest element is 3 and etc.

Challenge

O(n) time, O(1) extra memory.

Analysis

Sort Array

- O(nlogn) time

Max Heap

- O(nlogk) running time + O(k) memory

在数字集合中寻找第k大，可以考虑用Max Heap，将数组遍历一遍，加入到一个容量为k的PriorityQueue，最后poll() k-1次，那么最后剩下在堆顶的就是kth largest的数字了。

另外此题用quick sort中的 quick select 的思路来解，更优化，参考：http://www.jiuzhang.com/solutions/kth-largest-element/

Quick Select

- Time Complexity: average = O(n); worst case O(n^2), O(1) space

注意事项：

- partition的主要思想：将比pivot小的元素放到pivot左边，比pivot大的放到pivot右边
- pivot的选取决定了partition所得结果的效率，可以选择left pointer，更好的选择是在left和right范围内随机生成一个；

Time Complexity O(n)来自于O(n) + O(n/2) + O(n/4) + ... ~ O(2n)，此时每次partition的pivot大约将区间对半分。

Source: https://discuss.leetcode.com/topic/15256/4-c-solutions-using-partition-max-heap-priority_queue-and-multiset-respectively

> So, in the **average** sense, the problem is reduced to approximately half of its original size, giving the recursion `T(n) = T(n/2) + O(n)` in which `O(n)` is the time for partition. This recursion, once solved, gives `T(n) = O(n)` and thus we have a linear time solution. Note that since we only need to consider **one half** of the array, the time complexity is `O(n)`. If we need to consider both the two halves of the array, like quicksort, then the recursion will be `T(n) = 2T(n/2) + O(n)` and the complexity will be `O(nlogn)`.
>
> Of course, `O(n)` is the average time complexity. In the worst case, the recursion may become `T(n) = T(n - 1) + O(n)` and the complexity will be `O(n^2)`.

Solution

Sort Array

```
public class Solution {
    public int findKthLargest(int[] nums, int k) {
            final int N = nums.length;
            Arrays.sort(nums);
            return nums[N - k];
    }
}
```

Max Heap

```
class Solution {
    /*
     * @param k : description of k
     * @param nums : array of nums
     * @return: description of return
     */
    public int kthLargestElement(int k, int[] nums) {
        if (nums == null || nums.length == 0 || k == 0) {
            return -1;
        }
        PriorityQueue<Integer> heap = new PriorityQueue<Integer>
(k, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        for (int i = 0; i < nums.length; i++) {
            heap.offer(nums[i]);
        }
        for (int j = 0; j < k - 1; j++) {
            heap.poll();
        }
        return heap.peek();
    }
};
```

Quick Select (Partition with two pointers)

```
class Solution {
    /*
     * @param k : description of k
     * @param nums : array of nums
     * @return: description of return
     */
    public int kthLargestElement(int k, int[] nums) {
        if (nums == null || nums.length == 0 || k <= 0 || k > nu
ms.length) {
            return 0;
        }

        return select(nums, 0, nums.length - 1, nums.length - k)
;

    }

    public int select(int[] nums, int left, int right, int k) {
        if (left == right) {
            return nums[left];
        }

        int pivotIndex = partition(nums, left, right);
        if (pivotIndex == k) {
            return nums[pivotIndex];
        } else if (pivotIndex < k) {
            return select(nums, pivotIndex + 1, right, k);
        }  else {
            return select(nums, left, pivotIndex - 1, k);
        }
    }

    public int partition(int[] nums, int left, int right) {

        // Init pivot, better to be random
        int pivot = nums[left];
```

```
        // Begin partition
        while (left < right) {
            while (left < right && nums[right] >= pivot) { // sk
ip nums[i] that equals pivot
                right--;
            }
            nums[left] = nums[right];
            while (left < right && nums[left] <= pivot) { // ski
p nums[i] that equals pivot
                left++;
            }
            nums[right] = nums[left];
        }

        // Recover pivot to array
        nums[left] = pivot;
        return left;
    }
}
```

Quick Select (with random pivot)

Source: wikipedia: QuickSelect

Animation:

- geekviewpoint: quickselect
- csanimated: Quicksort

```
import java.util.Random;

class Solution {
    /*
     * @param k : description of k
     * @param nums : array of nums
     * @return: description of return
     */
    public int kthLargestElement(int k, int[] nums) {
        if (nums == null || nums.length == 0 || k <= 0 || k > nu
```

```
ms.length) {
            return 0;
        }

        return select(nums, 0, nums.length - 1, nums.length - k)
;

    }

    public int select(int[] nums, int left, int right, int k) {
        if (left == right) {
            return nums[left];
        }

        int pivotIndex = partition(nums, left, right);
        if (pivotIndex == k) {
            return nums[pivotIndex];
        } else if (pivotIndex < k) {
            return select(nums, pivotIndex + 1, right, k);
        }  else {
            return select(nums, left, pivotIndex - 1, k);
        }
    }

    public void swap(int[] nums, int x, int y) {
        int tmp = nums[x];
        nums[x] = nums[y];
        nums[y] = tmp;
    }

    public int partition(int[] nums, int left, int right) {

        Random rand = new Random();
        int pivotIndex = rand.nextInt((right - left) + 1) + left
;
        // Init pivot
        int pivotValue = nums[pivotIndex];

        swap(nums, pivotIndex, right);
```

```
        // First index that nums[firstIndex] > pivotValue
        int firstIndex = left;

        for (int i = left; i <= right - 1; i++) {
            if (nums[i] < pivotValue) {
                swap(nums, firstIndex, i);
                firstIndex++;
            }
        }

        // Recover pivot to array
        swap(nums, right, firstIndex);
        return firstIndex;
    }

    public static void main(String[] args) {
        System.out.println("kth Largest Element: Quick Select");
        int[] A = {21, 3, 34, 5, 13, 8, 2, 55, 1, 19};
        Solution search = new Solution();
        int expResult[] = {1, 2, 3, 5, 8, 13, 19, 21, 34, 55};
        int k = expResult.length;
        int err = 0;
        for (int exp : expResult) {
            if (exp != search.kthLargestElement(k--, A)) {
                System.out.println("Test failed: " + k);
                err++;
            }
        }
        System.out.println("Test finished");
    }
}
```

Reference

LeetCode:

- Solution explained
- Solutions using Partition, Max-Heap, priority_queue and multiset respectively

Source: wikipedia: QuickSelect

Animation:

- geekviewpoint: quickselect
- csanimated: Quicksort

Min Stack

Question

> Implement a stack with `min()` function, which will return the smallest number in the stack.
>
> It should support `push` , `pop` and `min` operation all in `O(1)` cost.
>
> *Notice*
>
> `min` operation will never be called if there is no number in the stack.
>
> **Example**
>
> ```
> push(1)
> pop()   // return 1
> push(2)
> push(3)
> min()   // return 2
> push(1)
> min()   // return 1
> ```

Analysis

利用两个栈结构，其中一个是主要的正常stack，满足pop(), push()的O(1)时间要求，另外一个作为辅助的minStack，仅存入min的integer。 min = Integer.parseInt(minStack.peek().toString());

push()时，如果number >= min，则push到minStack上 pop()时，如果number == min，也从minStack上pop

题中的例子，最终stack为[2, 3, 1], minStack为 [2, 1]

Solution

```
public class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<Integer>();
        minStack = new Stack<Integer>();
    }

    public void push(int number) {
        stack.push(number);
        if (minStack.isEmpty()) {
            minStack.push(number);
        } else if (Integer.parseInt(minStack.peek().toString())
>= number) {
            minStack.push(number);
        }
    }

    public int pop() {
        if (stack.peek().equals(minStack.peek())) {
            minStack.pop();
        }
        return stack.pop();
    }

    public int min() {
        return minStack.peek();
    }
}
```

Rehashing

Question

> The size of the hash table is not determinate at the very beginning. If the total size of keys is too large (e.g. size >= capacity / 10), we should double the size of the hash table and rehash every keys. Say you have a hash table looks like below:

`size=3, capacity=4`

```
[null, 21, 14, null]
       ↓    ↓
       9   null
       ↓
      null
```

The hash function is:

```
int hashcode(int key, int capacity) {
    return key % capacity;
}
```

here we have three numbers, 9, 14 and 21, where 21 and 9 share the same position as they all have the same hashcode 1 (21 % 4 = 9 % 4 = 1). We store them in the hash table by linked list.

rehashing this hash table, double the capacity, you will get:

`size=3, capacity=8`

```
index:   0    1    2    3     4    5    6   7
hash : [null, 9, null, null, null, 21, 14, null]
```

Given the original hash table, return the new hash table after rehashing .

Notice

For negative integer in hash table, the position can be calculated as follow:

- C++/Java: if you directly calculate -4 % 3 you will get -1. You can use function: a % b = (a % b + b) % b to make it is a non negative integer.
- Python: you can directly use -1 % 3, you will get 2 automatically.

Example

Given [null, 21->9->null, 14->null, null],

return [null, 9->null, null, null, null, 21->null, 14->null, null]

Analysis

此题的难度不大，只需要按照题目的要求实现代码就可以。不过需要注意的是：

1. C++/Java中，不能直接对负数使用取模运算，而需要用等式 `a % b = (a % b + b) % b` ，让所得到的hash值为非负数。
2. 所得到的新的HashTable中，可能依然存在碰撞，所以仍然需要在对应hashcode位置的ListNode tail上插入新的ListNode。

Solution

```
/**
 * Definition for ListNode
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    /**
```

```
     * @param hashTable: A list of The first node of linked list
     * @return: A list of The first node of linked list which ha
ve twice size
     */
    public ListNode[] rehashing(ListNode[] hashTable) {
        if (hashTable == null || hashTable.length == 0) {
            return hashTable;
        }
        int capacity = hashTable.length;
        int newCapacity = 2 * capacity;
        ListNode[] newHashTable = new ListNode[newCapacity];
        for (int i = 0; i < capacity; i++) {
            ListNode ln = hashTable[i];
            while (ln != null) {
                int code = hashcode(ln.val, newCapacity);
                insertToHashTable(newHashTable, code, ln.val);
                ln = ln.next;
            }
        }
        return newHashTable;
    }
    public int hashcode(int key, int capacity) {
        int hash;
        if (key < 0) {
            hash = (key % capacity + capacity) % capacity;
        } else {
            hash = key % capacity;
        }
        return hash;
    }
    private void insertToHashTable(ListNode[] hashTable, int cod
e, int value) {
        if (code < hashTable.length) {
            ListNode ln = hashTable[code];
            if (ln == null) {
                hashTable[code] = ln = new ListNode(value);
            } else {
                while (ln.next != null) {
                    ln = ln.next;
                }
```

```
                ln.next = new ListNode(value);
            }
        }
    }

    public static void main(String[] args) {
        Solution s = new Solution();
        ListNode[] lsn = new ListNode[3];
        lsn[0] = null;
        lsn[1] = null;
        lsn[2] = new ListNode(29);
        lsn[2].next = new ListNode(5);
        ListNode[] newLsn = s.rehashing(lsn);
    }
};

class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val = x;
        next = null;
    }
}
```

Implement Queue by Two Stacks

Question

As the title described, you should only use two stacks to implement a queue's actions.

The queue should support `push(element)`, `pop()` and `top()` where pop is pop the first(a.k.a front) element in the queue.

Both pop and top methods should return the value of first element.

Example

```
push(1)
pop()     // return 1
push(2)
push(3)
top()     // return 2
pop()     // return 2
```

Analysis

用两个Stack来实现一个Queue，可以考虑到push()时，几乎与Queue中的offer()一样，都是加在末尾，区别是当Stack pop()时，取出的是最近加入（newest）的元素，而Queue用poll()则是将最老（oldest）的元素取出。使用2个Stack，可以将stack2作为push()时的目标，而另一个stack1用来翻转顺序，只有当peek()或者是poll()时，才需要将元素翻转存入stack1，再进行读取。

Solution

```
public class Queue {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public Queue() {
        stack1 = new Stack<Integer>();
        stack2 = new Stack<Integer>();
    }

    private void switchStack() {
        while (!stack2.empty()) {
            stack1.push(stack2.pop());
        }
    }

    public void push(int element) {
        stack2.push(element);
    }

    public int pop() {
        if (stack1.empty()) {
            switchStack();
        }
        return stack1.pop();
    }

    public int top() {
        if (stack1.empty()) {
            switchStack();
        }
        return stack1.peek();
    }
}
```

Reference

- http://algs4.cs.princeton.edu/ QueueWithTwoStacks.java

Stack Sorting

Question

Sort a stack in ascending order (with biggest terms on top).

You may use at most one additional stack to hold items, but you may not copy the elements into any other data structure (e.g. array).

Example

Given stack =

```
| |
|3|
|1|
|2|
|4|
-
```

return:

```
| |
|4|
|3|
|2|
|1|
-
```

The data will be serialized to [4,2,1,3]. The last element is the element on the top of the stack.

Challenge

`O(n^2)` time is acceptable.

Analysis

根据题目提示，可以设想利用另一个stack作为临时储存空间，设定的规则是：

1. 从origin stack中不断pop() element
2. 对于helper stack，如果helper stack peek() < element，则将helper stack中的元素全部转移到origin stack
3. 再将element push()到helper stack中
4. 不断重复上述步骤，直到origin stack isEmpty
5. 最后，所有的元素已经按照descending order排序好（smallest on top），只需将其转移到origin stack，则origin stack即为所需排序

时间复杂度为O(n^2)，空间复杂度为O(n)

Solution

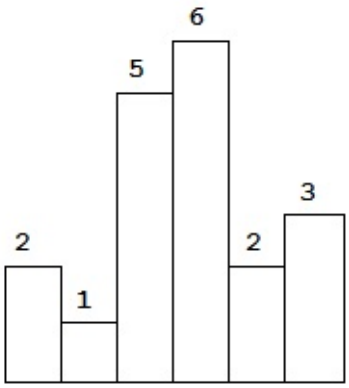
```
public class Solution {
    /**
     * @param stack an integer stack
     * @return void
     */
    public void stackSorting(Stack<Integer> stack) {
        Stack<Integer> helpStack = new Stack<Integer>();
        while (!stack.isEmpty()) {
            int element = stack.pop();
            while (!helpStack.isEmpty() && helpStack.peek() < el
ement) {
                stack.push(helpStack.pop());
            }
            helpStack.push(element);
        }
        while (!helpStack.isEmpty()) {
            stack.push(helpStack.pop());
        }
    }
}
```

Reference

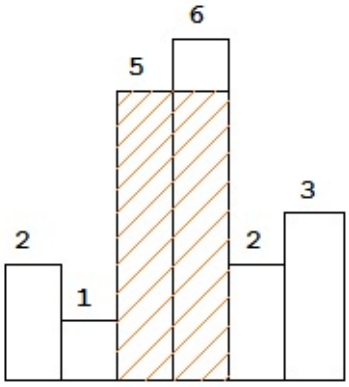
- IBM interview question : how to sort a stack?

Largest Rectangle in Histogram

Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].



The largest rectangle is shown in the shaded area, which has area = 10 unit.

Example

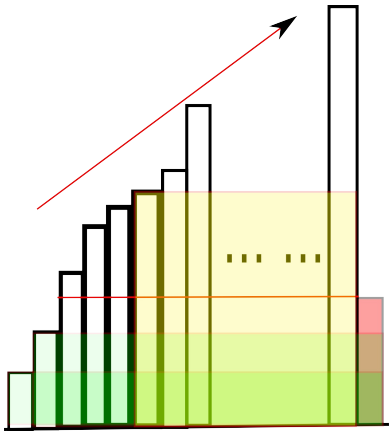
Given height = [2,1,5,6,2,3], return 10.

Analysis

最基本的解法就是两重循环，遍历所有[i, j]，寻找其中最矮bar，得出矩形面积，时间复杂度为O(n^2)，不过这样的解法会得到TLE; 一个简单的改进是，只对合适的右边界（峰顶），往左遍历面积，这个优化只是比较有效的剪枝，算法仍然是O(n^2)。

而此题最佳解法则是利用一个Stack，主要思想是维护一个单调递增的栈（栈内存元素的下标），比较栈顶（下标对应元素）与当前元素，如果当前元素大于栈顶（下标对应元素）则入栈，否则一直出栈，并逐个计算面积（取最大值），直到栈顶（下标对应元素）小于当前元素。也就是说栈内（下标对应元素）都大于等于当前元素。

如图所示，



图片来源：http://www.cnblogs.com/lichen782/p/leetcode_Largest_Rectangle_in_Histogram.html

需要注意的一点是，对原height[]的最后增加一位0，用来最终的i位置，确保所有的（下标对应的元素）都完成出栈（因为所有的元素都大于0）。

另附2003/2004 ACM University of Ulm Local Contest 题解之一：

> **Linear search using a stack of incomplete subproblems**
>
> We process the elements in left-to-right order and maintain a stack of information about started but yet unfinished subhistograms. Whenever a new element arrives it is subjected to the following rules. If the stack is empty we open a new subproblem by pushing the element onto the stack. Otherwise we compare it to the element on top of the stack. If the new one is greater we again push it. If the new one is equal we skip it. In all these cases, we continue with the next new element. If the new one is less, we finish the topmost subproblem by updating the maximum area w.r.t. the element at the top of the stack. Then, we discard the element at the top, and repeat the procedure keeping the current new element. This way, all subproblems are finished until the stack becomes empty, or its top element is less than or equal

> to the new element, leading to the actions described above. If all elements have been processed, and the stack is not yet empty, we finish the remaining subproblems by updating the maximum area w.r.t. to the elements at the top. For the update w.r.t. an element, we find the largest rectangle that includes that element. Observe that an update of the maximum area is carried out for all elements except for those skipped. If an element is skipped, however, it has the same largest rectangle as the element on top of the stack at that time that will be updated later. The height of the largest rectangle is, of course, the value of the element. At the time of the update, we know how far the largest rectangle extends to the right of the element, because then, for the first time, a new element with smaller height arrived. The information, how far the largest rectangle extends to the left of the element, is available if we store it on the stack, too. We therefore revise the procedure described above. If a new element is pushed immediately, either because the stack is empty or it is greater than the top element of the stack, the largest rectangle containing it extends to the left no farther than the current element. If it is pushed after several elements have been popped off the stack, because it is less than these elements, the largest rectangle containing it extends to the left as far as that of the most recently popped element. Every element is pushed and popped at most once and in every step of the procedure at least one element is pushed or popped. Since the amount of work for the decisions and the update is constant, the complexity of the algorithm is O(n) by amortized analysis.

Solution

Naive Implementation - TLE

```
public class Solution {
    /**
     * @param height: A list of integer
     * @return: The area of largest rectangle in the histogram
     */
    public int largestRectangleArea(int[] height) {
        int maxArea = 0;
        int[] min = new int[height.length];
        for (int i = 0; i < height.length; i++) {
            for (int j = i; j < height.length; j++) {
                if (i == j) {
                    min[j] = height[j];
                } else {
                    if (height[j] < min[j - 1]) {
                        min[j] = height[j];
                    } else {
                        min[j] = min[j - 1];
                    }
                }
                int tempArea = min[j] * (j - i + 1);
                if (tempArea > maxArea) {
                    maxArea = tempArea;
                }
            }
        }

        return maxArea;
    }
}
```

Pruning - AC

```
public class Solution {
    /**
    * @param height: A list of integer
    * @return: The area of largest rectangle in the histogram
    */
    public int largestRectangleArea(int[] height) {
        int maxV = 0;
        for(int i =0; i< height.length; i++)
        {
            if(i+1 < height.length && height[i] <= height[i+1])
{
                // if not peak node, skip it
                continue;
            }
            int minV = height[i];
            for(int j =i; j>=0; j--)
            {
                minV = Math.min(minV, height[j]);
                int area = minV*(i-j+1);
                if(area > maxV)
                maxV = area;
            }
        }
        return maxV;
    }
}
```

O(n) Linear search using a stack of incomplete subproblems

```
public class Solution {
    public int largestRectangleArea(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }

        Stack<Integer> stack = new Stack<Integer>();
        int max = 0;
        for (int i = 0; i <= height.length; i++) {
            int current = (i == height.length) ? -1 : height[i];
            while (!stack.isEmpty() && current <= height[stack.p
eek()]) {
                int h = height[stack.pop()];
                int w = stack.isEmpty() ? i : i - stack.peek() -
1;
                max = Math.max(max, h * w);
            }
            stack.push(i);
        }

        return max;
    }
}
```

Another Implementation of Linear search using a stack of incomplete subproblems

```
public class Solution {
    /**
     * @param height: A list of integer
     * @return: The area of largest rectangle in the histogram
     */
    public int largestRectangleArea(int[] height) {
        Stack<Integer> stack = new Stack<Integer>();
        int i = 0;
        int maxArea = 0;
        int[] h = new int[height.length + 1];
        // add an 0, so it would calculate the last height
        h = Arrays.copyOf(height, height.length + 1);
        while(i < h.length){
            if(stack.isEmpty() || h[stack.peek()] <= h[i]){
                stack.push(i++);
            }else {
                int t = stack.pop();
                maxArea = Math.max(maxArea, h[t] * (stack.isEmpt
y() ? i : i - stack.peek() - 1));
            }
        }
        return maxArea;
    }
}
```

Reference

- LeetCode 笔记系列 17 Largest Rectangle in Histogram
- Largest Rectangular Area in a Histogram
- Largest Rectangle in Histogram 解题报告
- Problem H: Largest Rectangle in a Histogram
- Problem H: Largest Rectangle in a Histogram - Judge
- LargestRectangleArea.java
- Largest Rectangle in Histogram O(n) 解法详析， Maximal Rectangle
- LeetCode: Largest Rectangle in Histogram(直方图最大面积)

Longest Consecutive Sequence

Question

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

Clarification

Your algorithm should run in O(n) complexity.

Example

Given [100, 4, 200, 1, 3, 2],

The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Analysis

主要有两种思路:

1. 对于每一个n，可以检查n-1, n+1是否存在于这个set（或者map）中；对于map的每个操作都是O(1)的，所以最终是O(n);
2. 对于每一个n，检查是否为一个consecutive sequence的边界，也就是n-1不存在于set中，再逐次检查n + 1, n + 2, n + 3...是否在set中，最终得到另一个上边界（+1）m，所以sequence的长度为m - n （也可以是n+1不存在于set中，则反向检查）。转为set，时间O(n)，之后对于set中的每一个元素，如果是一个连续序列的下边界，则对这个连续序列进行，因为对于每一个连续序列实际只会扫描一遍，所以这个循环最终是O(n)时间复杂度的。

另外，从题目对于时间复杂度的要求O(n)，可以推测那么解法可能是不可以是重循环。

对于第一种思路，具体的解释如下：https://leetcode.com/discuss/18886/my-really-simple-java-o-n-solution-accepted

Whenever a new element n is inserted into the map, do two things:

1. See if **n - 1** and **n + 1** exist in the map, and if so, it means there is an existing sequence next to **n**. Variables **left** and **right** will be the length of those two sequences, while 0 means there is no sequence and n will be the boundary point later. Store **(left + right + 1)** as the associated value to key n into the map.
2. Use **left** and **right** to locate the other end of the sequences to the left and right of n respectively, and replace the value with the new length.

Everything inside the for loop is O(1) so the total time is O(n)

第二种思路来源：https://leetcode.com/discuss/38619/simple-o-n-with-explanation-just-walk-each-streak

Solution

O(n) HashMap

```
public int longestConsecutive(int[] num) {
    int res = 0;
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer
>();
    for (int n : num) {
        if (!map.containsKey(n)) {
            int left = (map.containsKey(n - 1)) ? map.get(n - 1)
 : 0;
            int right = (map.containsKey(n + 1)) ? map.get(n + 1
) : 0;
            // sum: length of the sequence n is in
            int sum = left + right + 1;
            map.put(n, sum);

            // keep track of the max length
            res = Math.max(res, sum);

            // extend the length to the boundary(s)
            // of the sequence
            // will do nothing if n has no neighbors
            map.put(n - left, sum);
            map.put(n + right, sum);
        }
        else {
            // duplicates
            continue;
        }
    }
    return res;
}
```

O(n) Convert to set, loop lower bound consecutive sequence

```java
public class Solution {
    /**
     * @param nums: A list of integers
     * @return an integer
     */
    public int longestConsecutive(int[] nums) {
        // write you code here
        Set<Integer> hs = new HashSet<Integer>();
        for (int n : nums) {
            hs.add(n);
        }
        int longest = 0;
        for (int n : hs) {
            if (!hs.contains(n - 1)) {
                int m = n + 1;
                while (hs.contains(m)) {
                    m++;
                }
                longest = Math.max(longest, m - n);
            }
        }
        return longest;
    }
}
```

Animal Shelter

Question

An animal shelter holds only dogs and cats, and operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog and dequeueCat.

Example

```
int CAT = 0
int DOG = 1

enqueue("james", DOG);
enqueue("tom", DOG);
enqueue("mimi", CAT);
dequeueAny();  // should return "james"
dequeueCat();  // should return "mimi"
dequeueDog();  // should return "tom"
```

Challenge

Can you do it with single Queue?

Analysis

解法一

关于时间oldest优先，可以从两个角度考虑

1. Queue本身的first-in-first-out(FIFO)特性
2. 元素自带时间戳time stamp，方便比较

于是，这里可以存两个LinkedList：cats，dogs，链表中的元素除了name，还可以存入time，这样当dequeueAny()时，可以通过time来决定哪一个最先出队列。dequeue均可以在O(1)时间内实现。

challenge里所说的Single Queue的方法，难点在于shift queue，因为不能用两个queue分别来track同一个type中的先后顺序，那么就需要用shift的方式来维持队列。相应的dequeue算法复杂度也高很多O(n)

Solution

2 Queue

```
class Node {
    int time;
    String name;

    Node(String name, int time) {
        this.name = name;
        this.time = time;
    }
    public String getName() {
        return this.name;
    }
    public int getTime() {
        return this.time;
    }
}

public class AnimalShelter {

    private static final int DOG = 1;
    private static final int CAT = 0;

    private int tot;
    private LinkedList<Node> cats, dogs;

    public AnimalShelter() {
        // do initialize if necessary
```

```
        tot = 0;
        dogs = new LinkedList<Node>();
        cats = new LinkedList<Node>();
    }

    /**
     * @param name a string
     * @param type an integer, 1 if Animal is dog or 0
     * @return void
     */
    void enqueue(String name, int type) {
        tot += 1;
        if (type == DOG) {
            dogs.add(new Node(name, tot));
        } else {
            cats.add(new Node(name, tot));
        }
    }

    public String dequeueAny() {
        if (cats.isEmpty()) {
            return dequeueDog();
        } else if (dogs.isEmpty()) {
            return dequeueCat();
        } else {
            int dogTime = dogs.getFirst().getTime();
            int catTime = cats.getFirst().getTime();
            if (catTime < dogTime) {
                return dequeueCat();
            } else {
                return dequeueDog();
            }
        }
    }

    public String dequeueDog() {
        String name = dogs.getFirst().getName();
        dogs.removeFirst();
        return name;
    }
```

```
    public String dequeueCat() {
        String name = cats.getFirst().getName();
        cats.removeFirst();
        return name;
    }
}
```

Single Queue

```
class Node {
    String name;
    int type;
    Node(String name, int type) {
        this.name = name;
        this.type = type;
    }
}
public class AnimalShelter {
    Queue<Node> queue;
    public AnimalShelter() {
        // do initialize if necessary
        queue = new LinkedList<Node>();
    }

    /**
     * @param name a string
     * @param type an integer, 1 if Animal is dog or 0
     * @return void
     */
    void enqueue(String name, int type) {
        // Write your code here
        queue.offer(new Node(name, type));
    }

    public String dequeueAny() {
        // Write your code here
        Node node = queue.poll();
        return node.name;
```

```
    }

    public String dequeueDog() {
        // Write your code here
        return dequeueType(1);
    }

    public String dequeueCat() {
        // Write your code here
        return dequeueType(0);
    }

    private String dequeueType(int type) {
        int shiftTime = 0;
        while (queue.peek().type != type) {
            queue.offer(queue.poll());
            shiftTime++;
        }
        Node node = queue.poll();
        shiftTime = queue.size() - shiftTime;
        while (shiftTime != 0) {
            queue.offer(queue.poll());
            shiftTime--;
        }
        return node.name;
    }
}
```

Number of Airplanes in the Sky

Question

Given an interval list which are flying and landing time of the flight. How many airplanes are on the sky at most?

Notice

If landing and flying happens at the same time, we consider landing should happen at first.

Example

For interval list

```
[
  [1,10],
  [2,3],
  [5,8],
  [4,7]
]
```

Return `3`

Tags

LintCode Copyright Array Interval

Related Problems

Easy Merge Intervals

Analysis

HashMap - 空间换时间时间

航班起飞降落时间，直观的想法是，把一段段飞行时间看成线段，将它们都投射到时间坐标上，并进行叠加，最高的时间坐标点则是空中飞机最多的时候。 用什么来表示时间坐标呢？可以利用HashMap，记录每一个时间段里的每一个飞行时间点，这样可以方便累加；（更简单的，可以直接用一个Array，因为24小时的客观限制，int[24]。不过最好跟面试官明确一下时间坐标的可能范围，比如，是否会出现跨天的时间段，[18, 6]，也就是18:00PM - 6:00AM +1，如果有这种情况，则需要考虑进行适当的处理。不过根据OJ给出的test cases，只会出现end>start的情况，并且不受24小时的时间限制，因此使用HashMap更为合适）

Sweep Line - 扫描线法

可以对于各个飞行时间段按照start时间进行排序（附加start，end的flag，如果time相同时，end在start前）。那么遍历这个排序过的链表时，也就是相当于在时间线上从前向后顺序移动，遇到start就+1，遇到end就-1，记录其中的最大值max即可。

Solution

HashMap

```
/**
 * Definition of Interval:
 * public classs Interval {
 *     int start, end;
 *     Interval(int start, int end) {
 *         this.start = start;
 *         this.end = end;
 *     }
 */

class Solution {
    /**
     * @param intervals: An interval array
     * @return: Count of airplanes are in the sky.
     */
    public int countOfAirplanes(List<Interval> airplanes) {
        if (airplanes == null || airplanes.size() == 0) {
```

```
            return 0;
        }

        HashMap<Integer, Integer> hashmap = new HashMap<Integer,
 Integer>();
        int max = 0;

        for (Interval flight : airplanes) {
            int start = flight.start;
            int end = flight.end;
            for (int i = start; i < end; i++) {
                if (hashmap.containsKey(i)) {
                    hashmap.put(i, hashmap.get(i) + 1);
                } else {
                    hashmap.put(i, 1);
                }
                max = Math.max(max, hashmap.get(i));
            }
        }
        return max;
    }
}
```

Sweep Line

```
/**
 * Definition of Interval:
 * public class Interval {
 *     int start, end;
 *     Interval(int start, int end) {
 *         this.start = start;
 *         this.end = end;
 *     }
 */

class Point {
    int time;
    int delta;
```

```
    Point(int time, int delta) {
        this.time = time;
        this.delta = delta;
    }
}

public class Solution {
    /**
     * @param intervals: An interval array
     * @return: Count of airplanes are in the sky.
     */
    public int countOfAirplanes(List<Interval> airplanes) {
        if (airplanes == null || airplanes.size() == 0) {
            return 0;
        }
        List<Point> timePoints = new ArrayList<Point>(airplanes.
size() * 2);
        for (Interval flight : airplanes) {
            timePoints.add(new Point(flight.start, 1));
            timePoints.add(new Point(flight.end, -1));
        }

        // Sort the flight time intervals
        Collection.sort(timePoints, new Comparator<Point>() {
            public int compare(Point a, Point b) {
                if (a.time == b.time) {
                    return a.delta - b.delta;
                } else {
                    return a.time - b.time;
                }
            }
        });

        int max = 0;
        int sum = 0;

        // Go through the time points
        for (Point p : timePoints) {
            sum += p.delta;
```

```
                max = Math.max(sum, max);
            }

            return max;
        }
    }
```

Reference

- Jiuzhang
- feliciafay: LintCode Number of Airplanes in the Sky(Java)

Data Stream Median

Question

Numbers keep coming, return the median of numbers at every time a new number added.

Clarification

What's the definition of Median?

- Median is the number that in the middle of a sorted array. If there are n numbers in a sorted array A, the median is A[(n - 1) / 2]. For example, if A= [1,2,3], median is 2. If A=[1,19], median is 1.

Example

For numbers coming list: [1, 2, 3, 4, 5], return [1, 1, 2, 2, 3].

For numbers coming list: [4, 5, 1, 3, 2, 6, 0], return [4, 4, 4, 3, 3, 3, 3].

For numbers coming list: [2, 20, 100], return [2, 2, 20].

Challenge

Total run time in O(nlogn).

Tags

LintCode Copyright Heap Priority Queue Google

Related Problems

Hard Sliding Window Median 17 % Easy Median 22 % Hard Median of two Sorted Arrays

Analysis

寻找中位数median，这里有一种很巧妙的思路，需要利用Heap的半排序特性，是指root node的值是min （min heap）或者max（max heap）。

利用两个heap，一个minHeap，一个maxHeap，将nums[]的数顺序存入时，则可以分别存入maxHeap和minHeap，并保持这两个heap的size相同，保持两者size相同的操作通过 `minHeap.offer(maxHeap.poll());` 和 `maxHeap.offer(minHeap.poll());`

> The basic idea is to maintain two heaps: a max-heap and a min-heap. **The max heap stores the smaller half of all numbers while the min heap stores the larger half.** The sizes of two heaps need to be balanced each time when a new number is inserted so that their size will not be different by more than 1. Therefore each time when findMedian() is called we check if two heaps have the same size. If they do, we should return the average of the two top values of heaps. Otherwise we return the top of the heap which has one more element. -- @hanhanbu
>
> https://discuss.leetcode.com/topic/27506/easy-to-understand-double-heap-solution-in-java

Notice

LeetCode中与LintCode里稍有不同，在于对于Median的定义：

LeetCode要求是如果是even number，就计算中间两个数字的平均值，也就是 `(maxHeap.peek() + (minHeap.peek()))/2`

而LintCode的要求则是当even number时取[N/2]那一个，也就是说不论是否even number，需要返回的都是 `maxHeap.peek()`

延伸思考

这里建立Heap依然需要知道即将传入的nums[]的长度，如果对于一个未知长度的nums[]这里应当如何处理呢？

Solution

Double Heap (minHeap + maxHeap)

```
public class Solution {
    PriorityQueue<Integer> maxHeap;//lower half
    PriorityQueue<Integer> minHeap;//higher half
```

```
     /**
     * @param nums: A list of integers.
     * @return: the median of numbers
     */
    public int[] medianII(int[] nums) {

        int count = nums.length;
        maxHeap = new PriorityQueue<Integer>(count, Collections.
reverseOrder());
        minHeap = new PriorityQueue<Integer>(count);

        int[] ans = new int[count];

        for (int i = 0; i < count; ++i) {
            addNum(nums[i]);
            ans[i] = findMedian();
        }
        return ans;
    }

    // Adds a number into the data structure.
    public void addNum(int num) {
        maxHeap.offer(num);
        minHeap.offer(maxHeap.poll());

        if(maxHeap.size() < minHeap.size()){
            maxHeap.offer(minHeap.poll());
        }
    }

    // Returns the median of current data stream
    public int findMedian() {
        if(maxHeap.size() == minHeap.size()){
            return maxHeap.peek(); // Or `(maxHeap.peek() + (min
Heap.peek()))/2`
        }else{
            return maxHeap.peek();
        }
    }
}
```

Reference

- programcreek: LeetCode – Find Median from Data Stream (Java)
- Short simple Java/C++/Python, O(log n) + O(1)
- LeetCode Discussion: Easy to understand double-heap solution in Java

Sliding Window Maximum

Question

Given an array of n integer with duplicate number, and a moving window(size k), move the window at each iteration from the start of the array, find the maximum number inside the window at each moving.

Example

For array [1, 2, 7, 7, 8], moving window size k = 3. return [7, 7, 8]

At first the window is at the start of the array like this

[|1, 2, 7| ,7, 8] , return the maximum 7;

then the window move one step forward.

[1, |2, 7 ,7|, 8], return the maximum 7;

then the window move one step forward again.

[1, 2, |7, 7, 8|], return the maximum 8;

Challenge

o(n) time and O(k) memory

Tags

LintCode Copyright Deque Zenefits

Related Problems

Hard Sliding Window Matrix Maximum 35 % Hard Paint House II 28 % Hard Sliding Window Median

Analysis

开始想到用一个固定大小的Max Heap，但是由于Heap的删除操作比较麻烦，最好使用HashHeap，不过HashHeap在Java中并没有现成的implementation，而其实现十分繁琐，因此转而思考有没有别的数据结构。相比Sliding Window Median，这里寻找Maximum也许更容易一些，因为是一个局部极值，也许可以用stack或者queue来记录当前窗口的最大元素？但是单纯使用stack或者queue都不能很好地满足需要，因为想维护一个数据结构，能够保持其元素的单调递减性，其头部永远是当前window的maximum，如果有新的较大元素，则将该结构内比它小的元素都pop出来，再push新的较大元素。Java中恰好有这样一个数据结构：`Deque`，也就是double-ended-queue，“双端队列”之意，其中一个实现为`ArrayDeque`（参考：https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html），能够满足两端的offer, poll, peek操作。

对于example:

```
nums = [1, 2, 7, 7, 8]

k = 3
```

先将k-1个元素填入，

```
| 2 |
```

从kth元素开始，再依次加入新元素并且删除原有旧元素，保持sliding window的大小不变

```
Window:
[|1, 2, 7|, 7, 8]

Deque:
| 7 |

Output: [7]
```

```
Window:
[1, |2, 7, 7|, 8]

Deque:
| 7, 7 |

Output: [7, 7]
```

```
Window:
[1, 2, |7, 7, 8|]

Deque:
| 8 |

Output: [7, 7, 8]
```

对于每一个nums中的元素都只扫描一遍，在deque中的操作时间复杂度也是在O(k)数量级以下，因此总时间复杂度为O(n * k) ~ O(n)，空闲复杂度O(k)，用于维护一个Deque。

Solution

Deque

```
public class Solution {
    // Make sure the maximum number is at the head of the deque
    public void inQueue(Deque<Integer> deque, int num) {
        while (!deque.isEmpty() && deque.peekLast() < num) {
            deque.pollLast();
        }
        deque.offerLast(num);
    }

    // Remove the previous numbers for sliding window constraints

    public void outQueue(Deque<Integer> deque, int num) {
        if (deque.peekFist() == num) {
```

```
            deque.pollFist();
        }
    }
    /**
     * @param nums: A list of integers.
     * @return: The maximum number inside the window at each mov
ing.
     */
    public ArrayList<Integer> maxSlidingWindow(int[] nums, int k)
 {
        ArrayList<Integer> ans = new ArrayList<Integer>();
        Deque<Integer> deque = new ArrayDeque<Integer>();

        if (nums == null || nums.length == 0) {
            return ans;
        }

        // Initialize the deque with first k - 1 element
        for (int i = 0; i < k - 1; i++) {
            inQueue(deque, nums[i]);
        }

        // Continue from k-th element, for the maximum in each s
liding window
        for (int i = k - 1; i < nums.length; i++) {
            inQueue(deque, nums[i]);
            ans.add(deque.peekFirst());
            outQueue(deque, nums[i - k + 1]);
        }
    }
}
```

Reference

- LeetCode Articles:
- LeetCode Discussion: Java O(n) solution using deque with explanation
- Java Doc: ArrayDeque
- Tutorialspoint: Java ArrayDeque

Heap 操作

- 插入:将新元素放到heap[size+1]的位置每次比较它的它父亲元素,如果小于它的父亲,证明现在不满 足堆的性质,然后向上Sift Up
- 删除:将根节点和最后一个节点进行交换如果该节点大于其中一个儿子,那么将其与其较小的儿子进 行交换做Sift Down,直到该节点的儿子均大于它的值,或者它的儿子为空

Heap

接口 • O(logN) Push -> Sift Up • O(logN) Pop -> Sift Down • O(1) Top • O(N) Delete

HashHeap

```
// HashHeap Implementation

// 接口
// • O(logN) Push -> Sift Up
// • O(logN) Pop -> Sift Down • O(1) Top
// • O(logN) Delete

class HashHeap {
    ArrayList<Integer> heap;
    String mode;
    int size_t;
    HashMap<Integer, Node> hash;

    class Node {
        public Integer id;
        public Integer num;

        Node(Node now) {
            id = now.id;
            num = now.num;
```

```
        }

        Node(Integer first, Integer second) {
            this.id = first;
            this.num = second;
        }
    }

    public HashHeap(String mod) {
        // TODO Auto-generated constructor stub
        heap = new ArrayList<Integer>();
        mode = mod;
        hash = new HashMap<Integer, Node>();
        size_t = 0;
    }

    int peak() {
        return heap.get(0);
    }

    int size() {
        return size_t;
    }

    Boolean empty() {
        return (heap.size() == 0);
    }

    int parent(int id) {
        if (id == 0) {
            return -1;
        }
        return (id - 1) / 2;
    }

    int lson(int id) {
        return id * 2 + 1;
    }

    int rson(int id) {
```

```
        return id * 2 + 2;
    }

    boolean comparesmall(int a, int b) {
        if (a <= b) {
            if (mode == "min")
                return true;
            else
                return false;
        } else {
            if (mode == "min")
                return false;
            else
                return true;
        }

    }

    void swap(int idA, int idB) {
        int valA = heap.get(idA);
        int valB = heap.get(idB);

        int numA = hash.get(valA).num;
        int numB = hash.get(valB).num;
        hash.put(valB, new Node(idA, numB));
        hash.put(valA, new Node(idB, numA));
        heap.set(idA, valB);
        heap.set(idB, valA);
    }

    Integer poll() {
        size_t--;
        Integer now = heap.get(0);
        Node hashnow = hash.get(now);
        if (hashnow.num == 1) {
            swap(0, heap.size() - 1);
            hash.remove(now);
            heap.remove(heap.size() - 1);
            if (heap.size() > 0) {
                siftdown(0);
```

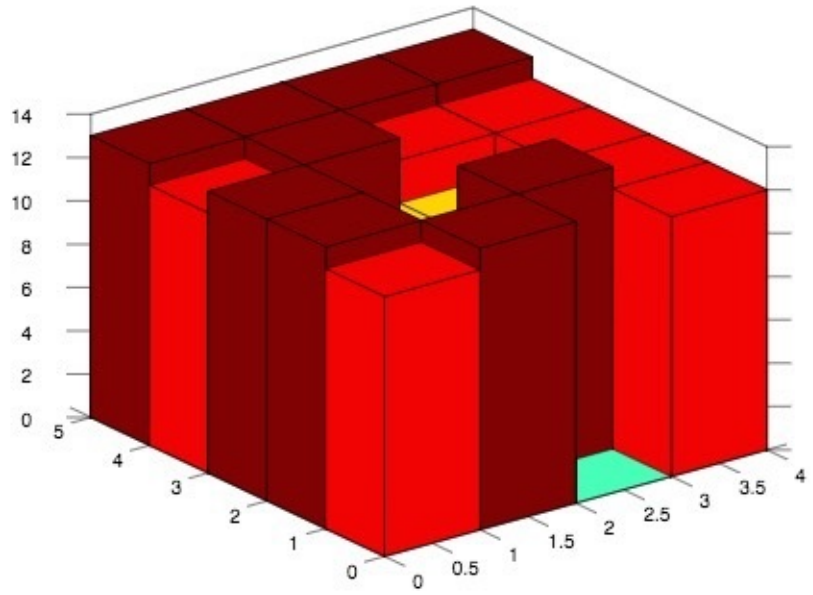
```
            }
        } else {
            hash.put(now, new Node(0, hashnow.num - 1));
        }
        return now;
    }

    void add(int now) {
        size_t++;
        if (hash.containsKey(now)) {
            Node hashnow = hash.get(now);
            hash.put(now, new Node(hashnow.id, hashnow.num + 1))
;

        } else {
            heap.add(now);
            hash.put(now, new Node(heap.size() - 1, 1));
        }

        siftup(heap.size() - 1);
    }

    void delete(int now) {
        size_t--;
        ;
        Node hashnow = hash.get(now);
        int id = hashnow.id;
        int num = hashnow.num;
        if (hashnow.num == 1) {

            swap(id, heap.size() - 1);
            hash.remove(now);
            heap.remove(heap.size() - 1);
            if (heap.size() > id) {
                siftup(id);
                siftdown(id);
            }
        } else {
            hash.put(now, new Node(id, num - 1));
        }
```

```
    }

    void siftup(int id) {
        while (parent(id) > -1) {
            int parentId = parent(id);
            if (comparesmall(heap.get(parentId), heap.get(id)) =
= true) {
                break;
            } else {
                swap(id, parentId);
            }
            id = parentId;
        }
    }

    void siftdown(int id) {
        while (lson(id) < heap.size()) {
            int leftId = lson(id);
            int rightId = rson(id);
            int son;
            if (rightId >= heap.size() || (comparesmall(heap.get
(leftId), heap.get(rightId)) == true)) {
                son = leftId;
            } else {
                son = rightId;
            }
            if (comparesmall(heap.get(id), heap.get(son)) == true
) {
                break;
            } else {
                swap(id, son);
            }
            id = son;
        }
    }
}
```

Trapping Rain Water II

Question

Given n x m non-negative integers representing an elevation map 2d where the area of each cell is 1 x 1, compute how much water it is able to trap after raining.



Example

Given 5*4 matrix

```
[12,13,0,12]
[13,4,13,12]
[13,8,10,12]
[12,13,12,12]
[13,13,13,13]
```

return `14` .

Tags

LintCode Copyright Heap Matrix

Related Problems

Medium Trapping Rain Water

Analysis

本题是Trapping Rain Water的follow up，I中是循环两遍记录每个位置左右两侧的最高水柱，而II在二维的灌水情境中，则需要从外围向内包围查找，记录最小的柱高，也就是木桶原理，最矮的柱子决定了灌水的高度。

1. 从最外围一圈向内部遍历，记录包围“墙”的最小柱高，可以利用min-heap（PriorityQueue）
2. 记录遍历过的点 `visited[][]`
3. 对于min-heap的堆顶元素，假设高度 `h` ，查找其周围4个方向上未曾访问过的点
   - 如果比 `h` 高，则说明不能装水，但是提高了“围墙”最低高度，因此将其加入min-heap中，设置元素被访问
   - 如果比 `h` 矮，则说明可以向其中灌水，且灌水高度就是 `h - h'` ，其中 `h'` 是当前访问的柱子高度，同样的，要将其加入min heap中，（且该元素高度记为灌水后的高度，也就是 `h` ，可以设想为一个虚拟的水位高度），设置元素被访问

此外，为了方便，可以定义一个Cell类，包含其坐标x,y，以及高度h，并定义其Comparator规则（也可以在初始化PriorityQueue的时候定义）。

Solution

```
class Cell {
    public int x, y, h;

    public Cell() {}

    public Cell(int x, int y, int h) {
        this.x = x;
```

```
        this.y = y;
        this.h = h;
    }
}

public class Solution {
    /**
     * @param heights: a matrix of integers
     * @return: an integer
     */
    public int trapRainWater(int[][] heights) {
        // Input validation
        if (heights == null || heights.length == 0 || heights[0]
.length == 0) {
            return 0;
        }

        int m = heights.length;
        int n = heights[0].length;

        // Initialize min-heap minheap, visited matrix visited[]
[]
        PriorityQueue<Cell> minheap = new PriorityQueue<Cell>(1,
new Comparator<Cell>() {
            public int compare(Cell c1, Cell c2) {
                if (c1.h > c2.h) {
                    return 1;
                } else if (c1.h < c2.h) {
                    return -1;
                } else {
                    return 0;
                }
            }
        });

        int[][] visited = new int[m][n];

        // Traverse the outer cells, add to the minheap
        for (int i = 0; i < m; i++) {
            minheap.offer(new Cell(i, 0, heights[i][0]));
```

```
            minheap.offer(new Cell(i, n - 1, heights[i][n - 1]))
;

            visited[i][0] = 1;
            visited[i][n - 1] = 1;
        }

        for (int j = 0; j < n; j++) {
            minheap.offer(new Cell(0, j, heights[0][j]));
            minheap.offer(new Cell(m - 1, j, heights[m - 1][j]))
;

            visited[0][j] = 1;
            visited[m - 1][j] = 1;
        }

        // Helper direction array
        int[] dirX = new int[] {0, 0, -1, 1};
        int[] dirY = new int[] {-1, 1, 0, 0};

        int water = 0;

        // Starting from the min height cell, check 4 direction
        while (!minheap.isEmpty()) {
            Cell now = minheap.poll();

            for (int k = 0; k < 4; k ++) {
                int x = now.x + dirX[k];
                int y = now.y + dirY[k];

                if (x < m && x >= 0 && y < n && y >= 0 && visite
d[x][y] != 1) {
                    minheap.offer(new Cell(x, y, Math.max(now.h,
 heights[x][y])));
                    visited[x][y] = 1;

                    // Fill in water or not
                    water += Math.max(0, now.h - heights[x][y]);
                }
            }
```

```
            }
            return water;
        }
    }
```

Reference

- Jason_Yuan: LintCode 364

Two Pointers

Summary

两个指针
- 对撞型 (2 sum 类 和 partition 类)
- 前向型 (窗口类, 快慢类)
- 两个数组,两个指针 (并行)

模板
- 2 Sum类模板
- Partition 类模板
- 窗口类模板

1. “对撞型”或“相会型”

Two Sum类题目思路

```
if (A[i] + A[j] > sum)
    j--;
    do something
else if (A[i] + A[j] < sum)
    i++;
    do something
else
    do something
    i++ or j--
```

灌水类型题目思路

```
if (A[i] > A[j])
    j--
else if (A[i] < A[j])
    i++
else
    i++ or j--
```

> 这一类通过对撞型指针优化算法，根本上其实要证明就是不用扫描多余状态

```
// Give an array arr[]
int left = 0;
int right = arr.length - 1;

while(left < right) {
    if(arr[left] 和 arr[right] 满足某一条件) {
        // Do something
        right --; // 不用考虑[left + 1, right - 1] 和 right 组成的pair
    } else if (arr[left] 和 arr[right] 不满足某一条件) {
        left ++; // 不用考虑[left + 1, right - 1] 和 left 组成的pair

    }
}
```

Partition类模板

```java
public int partition(int[] nums, int l, int r) {
    // 初始化左右指针和pivot
    int left = l, right = r;
    int pivot = nums[left];

    // 进行 partition
    while (left < right) {
        while (left < right && nums[right] >= pivot) {
            right--;
        }
        nums[left] = nums[right];
        while (left < right && nums[left] <= pivot) {
            left++;
        }
        nums[right] == nums[left];
    }

    // 返还pivot点到数组里
    nums[left] = pivot;
    return left;
}
```

Partition 另一种模板

```
public void swap(int[] nums, int x, int y) {
    int tmp = nums[x];
    nums[x] = nums[y];
    nums[y] = tmp;
}

public int partition(int[] nums, int left, int right) {

    Random rand = new Random();
    int pivotIndex = rand.nextInt((right - left) + 1) + left;
    // Init pivot
    int pivotValue = nums[pivotIndex];

    swap(nums, pivotIndex, right);

    // First index that nums[firstIndex] > pivotValue
    int firstIndex = left;

    for (int i = left; i <= right - 1; i++) {
        if (nums[i] < pivotValue) {
            swap(nums, firstIndex, i);
            firstIndex++;
        }
    }

    // Recover pivot to array
    swap(nums, right, firstIndex);
    return firstIndex;
}
```

- RosettaCode: Quickselect Algorithm
- Code Scream: The Partition Algorithm - Used For Quick-Select and Quick-Sort
- GeekViewpoint: Quickselect: Kth Greatest Value
- 被忽视的 partition 算法

相会型指针题目

- 2 Sum 类 （通过判断条件优化算法）
    - 3 Sum Closest
    - 4 Sum
    - 3 Sum
    - Two sum II
    - Triangle Count
    - Trapping Rain Water
    - Container With Most Water
- Partition 类
    - Partition-array
    - Sort Colors
    - Partition Array by Odd and Even
    - Sort Letters by Case
    - Valid Palindrome
    - quick sort\/ quick select\/ nuts bolts problem\/wiggle sort II

2. 前向型或者追击型

窗口类

窗口类指针移动模板

> 通过两层 `for` 循环改进算法 --- 不同于sliding window

```
优化类型:

* 优化思想通过两层for循环而来
* 外层指针依然是依次遍历
* 内层指针证明是否需要回退
```

```
for (i = 0; i < n; i++) {
    while (j < n) {
        if (满足条件)
            j++
            更行j状态
        else (不满足条件)
            break
    }
    更新i状态
}
```

快慢类

```
ListNode fast, slow;
fast = head.next;
slow = head;
while (fast != slow) {
    if(fast==null || fast.next==null)
        return false;
    fast = fast.next.next;
    slow = slow.next;
}
return true;
```

前向型指针题目

- 窗口类
    - Remove Nth Node From End of List
    - minimum-size-subarray-sum
    - Minimum Window Substring
    - Longest Substring with At Most K Distinct Characters
    - Longest Substring Without Repeating Characters
- 快慢类
    - Find the Middle of Linked L

两个数组两个指针

两个数组各找一个元素, 使得和等于target

1. 找一种
2. 找全部种类

Two Sum II

Question

Given an array of integers, find how many pairs in the array such that their sum is bigger than a specific target number. Please return the number of pairs.

Example

Given numbers = `[2, 7, 11, 15]`, target = 24. Return 1. (11 + 15 is the only pair)

Challenge

Do it in O(1) extra space and O(nlogn) time.

Tags

Two Pointers Sort

Analysis

与two sum问题相比，因为是“大于”的关系比较，因此用Hashmap不能达到记忆化查找的效果。这里用two pointers的思路求解则十分简明：

1. 先对数组排序，时间复杂度为O(nlogn)
2. 初始化left = 0, right = nums.length - 1，如果有一个满足nums[left] + nums[right] > target, 那么对于left ~ right - 1，都会满足条件，因此计入结果中，count = count + right - left。循环结束的条件是 left < right 不被满足，说明两个指针相遇，所有的情况都被遍历。这一个过程的时间复杂度是 O(n)。

Solution

```java
public class Solution {
    /**
     * @param nums: an array of integer
     * @param target: an integer
     * @return: an integer
     */
    public int twoSum2(int[] nums, int target) {
        // Invalid input or exception
        if (nums == null || nums.length == 0) {
            return 0;
        }

        // Sort the Array nums[] to ascending order
        Arrays.sort(nums);

        // Use two pointers
        int count = 0;
        int pl = 0;
        int pr = nums.length - 1;

        while (pl < pr) {
            if (nums[pl] + nums[pr] > target) {

                // Index from pl to pr - 1 will all be counted
                count = count + (pr - pl);
                pr--;
            } else {
                pl++;
            }
        }
        return count;
    }
}
```

Triangle Count

Question

Given an array of integers, how many three numbers can be found in the array, so that we can build an triangle whose three edges length is the three numbers that we find?

Example

Given array S = [3,4,6,7], return 3. They are:

```
[3,4,6]
[3,6,7]
[4,6,7]
```

Given array S = [4,4,4,4], return 4. They are:

```
[4(1),4(2),4(3)]
[4(1),4(2),4(4)]
[4(1),4(3),4(4)]
[4(2),4(3),4(4)]
```

Analysis

直白的想法是三重循环，i, j, k, 只要满足S[i] + S[j] > S[k]，或者S[i] + S[j] > S[k], 或者S[j] + S[k] > S[i]，该组合就计入总数。 不过显然，这种算法复杂度较高，为O(n^3)。

可以对问题进行转化：

1. 对数组排序，按照O(nlogn)计
2. 对数组下标循环，则内部转化为一个two sum II问题，即寻找 S[j] + S[k] > S[i] 有多少组，因为数组已排序，则可以使用two pointers的方法
3. 对于每一个i，初始化left = 0, right = i - 1，如果有一个满足S[left] + S[right] >

S[i], 那么对于left ~ right - 1 同样也满足，因此计入right-left到最终count中

以此类推，最终算法复杂度为O(n^2 + nlogn)

Solution

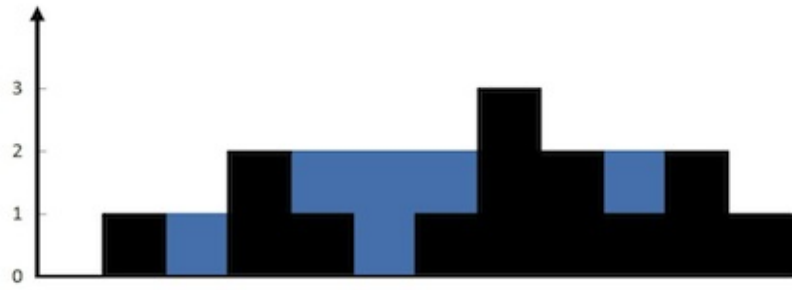
```
public class Solution {
    /**
     * @param S: A list of integers
     * @return: An integer
     */
    public int triangleCount(int S[]) {
        Arrays.sort(S);
        int count = 0;
        for (int i = 0; i < S.length; i++) {
            int left = 0;
            int right = i - 1;
            while (left < right) {
                if (S[left] + S[right] > S[i]) {
                    // The edge from S[left] to S[right - 1] wil
l also form a triangle
                    count += right - left;
                    right--;
                } else {
                    left++;
                }
            }
        }
        return count;
    }
}
```

Trapping Rain Water

Question

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it is able to trap after raining.

Trapping Rain Water



Example

Given `[0,1,0,2,1,0,1,3,2,1,2,1]`, return `6`.

Challenge

O(n) time and O(1) memory

O(n) time and O(n) memory is also acceptable.

Tags

Two Pointers Forward-Backward Traversal Array

Related Problems

- Medium Container With Most Water
- Hard Trapping Rain Water II
- Candy

Analysis

对于每一个bar来说，能装水的容量取决于左右两侧bar的最大值。扫描两次，一次从左向右，记录对于每一个bar来说其左侧的bar的最大高度left[i]，一次从右向左，记录每一个bar右侧bar的最大高度right[i]。第三次扫描，则对于每一个bar，计算（1）左侧最大height和当前bar的height的差值（left[i] - heights[i]）（2）右侧最大height和当前bar的height的差值（right[i] - heights[i]），取（1），（2）中结果小的那个作为当前bar的蓄水量。最终求和得到总蓄水量。

The water each bar can trap depends on the maximum height on its left and right. Thus scan twice - from left to right, and right to left and record the max height in each direction. The third time calculate the min difference between left/right height and current bar height. Sum the trapped water to get the final result.

Solution

```
public class Solution {
    /**
     * @param heights: an array of integers
     * @return: a integer
     */
    public int trapRainWater(int[] heights) {
        if (heights == null || heights.length == 0) {
            return 0;
        }
        int length = heights.length;

        int[] left = new int[length];
        int[] right = new int[length];

        int trapped = 0;

        // For heights[0] or heights[length - 1], the max left/r
ight height is 0
        left[0] = 0;
        right[length - 1] = 0;

        // Keep track of the max height on the left of height[i]
        for (int i = 1; i < length; i++) {
            left[i] = Math.max(left[i - 1], heights[i - 1]);
```

```
        }

        // Keep track of the max height on the right of height[i]

        for (int j = length - 2; j >= 0; j--) {
            right[j] = Math.max(right[j + 1], heights[j + 1]);
        }

        // Calculate the total trapped water
        for (int k = 0; k < length; k++) {
            if (Math.min(left[k], right[k]) - heights[k] > 0) {
                trapped +=  Math.min(left[k], right[k]) - height
s[k];

            }
        }

        return trapped;
    }
}
```

Reference

- Trapping Rain Water - GeeksforGeeks
- LeetCode – Trapping Rain Water (Java) - Program Creek
- 喜刷刷: LeetCode - Trapping Rain Water
- Yu's Garden LeetCode Question 111: Trapping Rain Water
- 水中的鱼：Trapping Rain Water 解题报告

Container With Most Water

Question

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate `(i, ai)`. n vertical lines are drawn such that the two endpoints of line i is at `(i, ai)` and `(i, 0)`. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Notice

You may not slant the container.

Example

Given [1,3,2], the max area of the container is 2.

Tags

Two Pointers Array

Related Problems

Medium Trapping Rain Water

Analysis

求灌水最多的两个柱子组合，与Trapping Rain Water有一些类似。联想到使用Two Pointers的方法。

two pointers方法需要确定

1. 何时两个pointers终止移动跳出循环，能够包含所有情况。这里仍然是left&right pointers相遇。
2. 何时移动左右pointers。灌水多少不仅与两个柱子的高度有关，也与两个柱子的距离有关。如何使得存放的水更多呢？较短的那一个柱子决定了灌水的area，因此可以移动指向较短的柱子的pointer，对于相会的pointer，一般趋势均为向中间移动，因此如果 `heights[left] < heights[right]` ，

则 `left++` ，反之 `right--` 。

比如，对于如下一个数组heights[]：

```
[1, 3, 1, 2, 4, 2]
```

应该如何移动pointers？

```
1 3 1 2 4 2
-----------
        |
  |     |
  |   | | |
| | | | | |
-----------
0 1 2 3 4 5
```

初始情况下， `left = 0` , `right = 5` , `heights[0] = 1` , `heights[5] = 2` ，也就是 `area = 5 * 1 = 5` ；因为 `heights[left] < heights[right]` ，因此 `left++` , `left = 1` ，更新后 `area = 4 * 2 = 8` ；这时， `heights[left] > heigths[right]` ，于是 `right--` , `right = 4` , 更新后 `area = 3 * 3 = 9` ；依次类推，当 `left` ， `right` 相遇时，循环结束，得到最大area为 `9` .

问题：为何较短的*pointer*向中间移动？

可以假设两个柱子距离为 `n` , 较短的柱子高度为 `h` ，那么中间可以存放水的area为 `n * h` 。假设初始状态下 `heights[left] < heights[right]` 。

如果令 `left++` ，此时柱子距离为 `n - 1` ，同时较短柱子为 `h1` ，那么 `area1 = (n - 1) * h1` ；如果令 `right--` ，此时柱子距离 `n - 1` ，同时较短柱子为 `h2` ，那么 `area2 = (n - 1) * h2` ；

两个方向得到的面积之差：

```
area1 - area2 = (n - 1) * h1 - (n - 1) * h2 = (n - 1) * (h1 - h2)
```

假定 `n - 1 >= 1` ，（n - 1 < 0 说明heights[]为空，n - 1 < 1说明 n = 1，则仅有一种area，不必讨论）

做一下不等式变换

```
area1 - area2 >= h1 - h2
```

因为 `heights[left] < heights[right]` 的设定，（并只考虑left/right中间的柱子高于两侧的情况，因为如果低于 `heights[left]`，`heights[right]` 将无法得到更大的水面积）如果 `right--` ，水位的最高高度将小于 `left++` 时，水位的最高高度。

Solution

Two Pointers

```java
public class Solution {
    /**
     * @param heights: an array of integers
     * @return: an integer
     */
    public int maxArea(int[] heights) {
        if (heights == null || heights.length == 0) {
            return 0;
        }

        int max = Integer.MIN_VALUE;

        int left = 0;
        int right = heights.length - 1;

        int area = 0;

        while (left < right) {
            if (heights[left] < heights[right]) {
                area = heights[left] * (right - left);
                max = Math.max(area, max);
                left++;
            } else {
                area = heights[right] * (right - left);
                max = Math.max(area, max);
                right--;
            }
        }
        return max;
    }
}
```

Two Pointers (Updated)

```java
public class Solution {
    /**
     * @param heights: an array of integers
     * @return: an integer
     */
```

```
    public int maxArea(int[] heights) {
        if (heights == null || heights.length == 0) {
            return 0;
        }

        int max = Integer.MIN_VALUE;

        int left = 0;
        int right = heights.length - 1;

        int area = 0;

        while (left < right) {
            area = calculateArea(left, right, heights);

            if (heights[left] < heights[right]) {
                max = Math.max(max, area);
                left++;
            } else {
                max = Math.max(max, area);
                right--;
            }
        }
        return max;
    }

    /**
     * Calculate area based on left, right pointers
     *
     * @param {int} left
     * @param {int} right
     * @param {int[]} heights
     * @return {int} an integer
     */
    public int calculateArea(int left, int right, int[] heights)
{
        return (right - left) * Math.min(heights[left], heights[
right]);
    }
}
```

Minimum Window Substring

Question

Given a string source and a string target, find the minimum window in source which will contain all the characters in target.

Notice

If there is no such window in source that covers all characters in target, return the empty string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in source.

Clarification

Should the characters in minimum window has the same order in target?

Not necessary.

Example

For source = `"ADOBECODEBANC"`, target = `"ABC"`, the minimum window is `"BANC"`

Challenge

Can you do it in time complexity O(n) ?

Tags

LinkedIn Hash Table Facebook

Analysis

可以用窗口型两个指针的思路来解决，外层for循环i = 0 ... n, 内层while循环，条件是 `j < source.length()` 和 `!isValid(sourceHash, targetHash)`，前者是数组下标边界，后者是一个函数，检查当前窗口中的substring是否包含了目标字符

串中全部所需的字符，未满足则继续扩大窗口j++，同时更新sourceHash。这里sourceHash，targetHash是整型数组int[]，因为ASCII码最多256个，因此用int[]可以作为简化的HashMap使用，key是char对应的int的值，value则是该char在substring中的出现个数。isValid函数则检查sourceHash是否全部包含了targetHash，256个字符一一对应，因此只需一重for循环，如果 `sourceHash[i] < targetHash[i]` ，则说明有字符未满足条件。

需要注意的是，要设定一个辅助变量记录*minStr*的长度。

Solution

```
public class Solution {

    /**
     * @param source: A string
     * @param target: A string
     * @return: A string denote the minimum window
     *          Return "" if there is no such a string
     */
    public String minWindow(String source, String target) {
        if (source == null || source.length() == 0 ||
                target == null || target.length() == 0) {
            return "";
        }
        int[] sourceHash = new int[256];
        int[] targetHash = new int[256];
        int ans = Integer.MAX_VALUE;
        String minStr = "";

        initTargetHash(targetHash, target);

        int i = 0;
        int j = 0;

        for (i = 0; i < source.length(); i++) {
            while (j < source.length() && !isValid(sourceHash, t
argetHash)) {
                sourceHash[source.charAt(j)]++;
```

```
                if (j < source.length()) {
                    j++;
                } else {
                    break;
                }
            }
            if (isValid(sourceHash, targetHash)) {
                if (ans > j - i) {
                    ans = Math.min(ans, j - i);
                    minStr = source.substring(i, j);
                }
            }
            sourceHash[source.charAt(i)]--;
        }
        return minStr;
    }
    boolean isValid(int[] sourceHash, int[] targetHash) {
        for (int i = 0; i < sourceHash.length; i++) {
            if (targetHash[i] > sourceHash[i]) {
                return false;
            }
        }
        return true;
    }

    public void initTargetHash(int[] targetHash, String target)
{
        for (int i = 0; i < target.length(); i++) {
            targetHash[target.charAt(i)]++;
        }
    }
}
```

Longest Substring Without Repeating Characters

Question

Given a string, find the length of the longest substring without repeating characters.

Example

For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3.

For "bbbbb" the longest substring is "b", with the length of 1.

Challenge

O(n) time

Tags

String Two Pointers Hash Table

Related Problems

Medium Longest Substring with At Most K Distinct Characters

Analysis

同样可以用窗口类型Two Pointers来解决，从两重for循环中优化算法，外层指针依次遍历，内层指针根据条件决定是否需要前进。

这样一类问题要熟练掌握思路：

```
for (i = 0; i < n; i++) {
    while (j < n) {
        if (满足条件)
            j++
            更行j状态
        else (不满足条件)
            break
    }
    更新i状态
}
```

Solution

```
public class Solution {
    /**
     * @param s: a string
     * @return: an integer
     */
    public int lengthOfLongestSubstring(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }
        HashSet<Character> hs = new HashSet<Character>();
        int ans = 0;

        int i = 0;
        int j = 0;
        for (i = 0; i < s.length(); i++) {
            while (j < s.length() && !hs.contains(s.charAt(j)))
{
                hs.add(s.charAt(j));
                ans = Math.max(ans, j - i + 1);
                j++;
            }

            hs.remove(s.charAt(i));
        }
        return ans;
    }
}
```

Reference

- Jiuzhang

Longest Substring with At Most K Distinct Characters

Question

Given a string s, find the length of the longest substring T that contains at most k distinct characters.

Example

For example, Given s = `"eceba"`, `k = 3`,

T is `"eceb"` which its length is `4`.

Challenge

O(n), n is the size of the string s.

Tags

String Two Pointers LintCode Copyright Hash Table

Related Problems

Medium Longest Substring Without Repeating Characters

Analysis

窗口型two pointers的又一实例。

需要注意的是内层while循环中的跳出条件，是 `!map.containsKey(ch_j) && map.size() == k` ，表明在目前出现了第一个character将使得 `map.size() > k` ，因此break。跳出内层while循环时， `j - i` 就是substring的长度，因为j指向了substring的下一个char (假设j指向substring的尾部字符，则应该用j - i + 1计算substring长度)。

Solution

```
public class Solution {
    /**
     * @param s : A string
     * @return : The length of the longest substring
     *           that contains at most k distinct characters.
     */
    public int lengthOfLongestSubstringKDistinct(String s, int k)
 {
        if (s == null || s.length() == 0 || k == 0) {
            return 0;
        }
        HashMap<Character, Integer> map = new HashMap<Character,
 Integer>();
        int strLength = s.length();
        int i = 0;
        int j = 0;
        int ans = Integer.MIN_VALUE;

        for (i = 0; i < strLength; i++) {
            while (j < strLength) {
                char ch_j = s.charAt(j);
                if (!map.containsKey(ch_j)) {
                    if (map.size() == k) {
                        break;
                    }
                    map.put(ch_j, 1);
                } else {
                    map.put(ch_j, map.get(ch_j) + 1);
                }
                j++;
            }
            ans = Math.max(ans, j - i);
            char ch_i = s.charAt(i);
            if (map.containsKey(ch_i)) {
                if (map.get(ch_i) - 1 == 0) {
                    map.remove(ch_i);
                } else {
                    map.put(ch_i, map.get(ch_i) - 1);
                }
```

```
                }

            }

            return ans;
        }
    }
```

Nuts & Bolts Problem

Question

Given a set of n nuts of different sizes and n bolts of different sizes. There is a one-one mapping between nuts and bolts. Comparison of a nut to another nut or a bolt to another bolt is not allowed. It means nut can only be compared with bolt and bolt can only be compared with nut to see which one is bigger/smaller.

We will give you a compare function to compare nut with bolt.

Example

Given nuts = ['ab','bc','dd','gg'], bolts = ['AB','GG', 'DD', 'BC'].

Your code should find the matching bolts and nuts.

one of the possible return:

nuts = ['ab','bc','dd','gg'], bolts = ['AB','BC','DD','GG'].

we will tell you the match compare function. If we give you another compare function.

the possible return is the following:

nuts = ['ab','bc','dd','gg'], bolts = ['BC','AA','DD','GG'].

So you must use the compare function that we give to do the sorting.

The order of the nuts or bolts does not matter. You just need to find the matching bolt for each nut.

Tags

Quick Sort Sort

Related Problems

Medium First Bad Version

Analysis

螺帽螺母问题脱胎于排序问题，这里的特别之处在于需要通过两个array进行对应的排序。这就需要利用一个array中的元素对另一个array进行partition，并反过来重复这一个过程，最终让两个array都满足comparator所定义的相同顺序。

这里要注意的是partition的变式，因为pivot并非来源当前array，只能通过一方元素作为基准，对另一个array进行partition。

> 核心在于：首先使用 nuts 中的某一个元素作为基准对 bolts 进行 partition 操作，随后将 bolts 中得到的基准元素作为基准对 nuts 进行 partition 操作。

Solution

Two Pointers Partition

http://www.jiuzhang.com/solutions/nuts-bolts-problem/

```
public class Solution {
    /**
     * @param nuts: an array of integers
     * @param bolts: an array of integers
     * @param compare: a instance of Comparator
     * @return: nothing
     */
    public void sortNutsAndBolts(String[] nuts, String[] bolts,
NBComparator compare) {
        if (nuts == null || bolts == null) return;
        if (nuts.length != bolts.length) return;

        qsort(nuts, bolts, compare, 0, nuts.length - 1);
    }

    private void qsort(String[] nuts, String[] bolts, NBComparat
or compare,
                       int l, int u) {
        if (l >= u) return;
        // find the partition index for nuts with bolts[l]
```

```
        int part_inx = partition(nuts, bolts[l], compare, l, u);
        // partition bolts with nuts[part_inx]
        partition(bolts, nuts[part_inx], compare, l, u);
        // qsort recursively
        qsort(nuts, bolts, compare, l, part_inx - 1);
        qsort(nuts, bolts, compare, part_inx + 1, u);
    }

    private int partition(String[] str, String pivot, NBComparat
or compare,
                          int l, int u) {
        for (int i = l; i <= u; i++) {
            if (compare.cmp(str[i], pivot) == 0 ||
                compare.cmp(pivot, str[i]) == 0) {
                swap(str, i, l);
                break;
            }
        }
        String now = str[l];
        int left = l;
        int right = u;
        while (left < right) {
            while (left < right &&
            (compare.cmp(str[right], pivot) == -1 ||
            compare.cmp(pivot, str[right]) == 1)) {
                right--;
            }
            str[left] = str[right];

            while (left < right &&
            (compare.cmp(str[left], pivot) == 1 ||
            compare.cmp(pivot, str[left]) == -1)) {
                left++;
            }
            str[right] = str[left];
        }
        str[left] = now;

        return left;
    }
```

```
    private void swap(String[] str, int l, int r) {
        String temp = str[l];
        str[l] = str[r];
        str[r] = temp;
    }
}
```

Another Way of Partition

http://algorithm.yuanbin.me/zh-hans/problem_misc/nuts_and_bolts_problem.html

```
/**
 * public class NBCompare {
 *     public int cmp(String a, String b);
 * }
 * You can use compare.cmp(a, b) to compare nuts "a" and bolts "
b",
 * if "a" is bigger than "b", it will return 1, else if they are
 equal,
 * it will return 0, else if "a" is smaller than "b", it will re
turn -1.
 * When "a" is not a nut or "b" is not a bolt, it will return 2,
 which is not valid.
*/
public class Solution {
    /**
     * @param nuts: an array of integers
     * @param bolts: an array of integers
     * @param compare: a instance of Comparator
     * @return: nothing
     */
    public void sortNutsAndBolts(String[] nuts, String[] bolts,
NBComparator compare) {
        if (nuts == null || bolts == null) {
            return;
        }
        if (nuts.length != bolts.length) {
            return;
        }
```

```
        int totalLength = nuts.length;
        qsort(nuts, bolts, 0, totalLength - 1, compare);
    }

    public void qsort(String[] nuts, String[] bolts, int l, int
r, NBComparator compare) {
        if (l >= r) {
            return;
        }
        // Find partition index for nuts, with bolts[l]
        int partIndex = partition(nuts, bolts[l], l, r, compare)
;
        // Partition bolts, with nuts[partIndex]
        partition(bolts, nuts[partIndex], l, r, compare);

        // quick sort recursively
        qsort(nuts, bolts, l, partIndex - 1, compare);
        qsort(nuts, bolts, partIndex + 1, r, compare);
    }

    public int partition(String[] arr, String pivot, int l, int
r, NBComparator compare) {
        // pivot is from another array
        int m = l;
        for (int i = l + 1; i <= r; i++) {
            if (compare.cmp(arr[i], pivot) == -1 || compare.cmp(
pivot, arr[i]) == 1) {
                m++;
                swap(arr, i, m);
            } else if (compare.cmp(arr[i], pivot) == 0 || compar
e.cmp(pivot, arr[i]) == 0) {
                swap(arr, i, l);
                i--;
            }
        }
        swap(arr, m, l);

        return m;
    }
```

```
    public void swap(String[] arr, int l, int r) {
        String temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
    }
}
```

Reference

- geeksforgeeks: Nuts & Bolts Problem (Lock & Key problem)
- yuanbin: Nuts and Bolts Problem

Valid Palindrome

Question

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Notice

Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

Example

`"A man, a plan, a canal: Panama"` is a palindrome.

`"race a car"` is not a palindrome.

Challenge

O(n) time without extra memory.

Tags

String Two Pointers Facebook Zenefits Uber

Related Problems

Medium Palindrome Linked List Medium Longest Palindromic Substring

Analysis

根据Palindrome本身对称性的特点，比较适合用two pointers的方法来解决，因为这里对valid palindrome的定义不考虑非字母和数字的字符，就需要在移动pointers时候注意条件判断，除了下标的边界，还有 `isAlphanumeric()` 来判定是否为字

母和数字。isAlphanumeric()可以用内置函数 `Character.isLetter()` 和 `Character.isDigit()` ，也可以根据 `char` ASCII码的定义写出。

Test Cases需要考虑周到，比如：

`" "` `"a..."` `"...a"` `"a.a"`

Solution

```
public class Solution {
    /**
     * @param s A string
     * @return Whether the string is a valid palindrome
     */
    public boolean isPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return true;
        }
        int i = 0;
        int j = s.length() - 1;

        while (i < j) {
            while (i < s.length() && !isAlphanumeric(s.charAt(i)
)) {
                i++;
            }
            if (i == s.length()) {
                return true;
            }
            while (j >= 0 && !isAlphanumeric(s.charAt(j))) {
                j--;
            }

            if (Character.toLowerCase(s.charAt(i)) != Character.
toLowerCase(s.charAt(j))) {
                break;
            } else {
                i++;
```

```
                j--;
            }
        }
        return (j <= i);
    }

    public boolean isAlphanumeric(char ch) {
        // return Character.isLetter(ch) || Character.isDigit(ch
);
        boolean result = (ch <= 'z' && ch >= 'a') ||
            (ch <= 'Z' && ch >= 'A') ||
            (ch <= '9' && ch >= '0');
        return result;

    }
}
```

Another implementation of isAlphanumeric()

```
    // Use Character methods isLetter(), isDigit()
    private boolean isAlphanumeric (char c) {
        return Character.isLetter(c) || Character.isDigit(c);
    }
```

Reference

http://www.jiuzhang.com/solutions/valid-palindrome/

The Smallest Difference

Question

Given two array of integers(the first array is array A, the second array is array B), now we are going to find a element in array A which is A[i], and another element in array B which is B[j], so that the difference between A[i] and B[j] (|A[i] - B[j]|) is as small as possible, return their smallest difference.

Example

For example, given array A = [3,6,7,4], B = [2,8,9,3], return 0

Challenge

O(n log n) time

Tags

Two Pointers LintCode Copyright Sort Array

Analysis

Brutal Force 双重for循环，找Math.abs(A[i] - B[j])的最小值

先排序 O(nlogn) 之后也有两种思路：

1. 再通过二分查找O(logn)，在B[j]中找接近A[i]的元素，找n次，总共时间复杂度O(n * logn)
2. two pointers，只要A[i] < B[j]，那么i++；反之，j++，循环结束条件就是i, j有一者达到数组边界

例如题目中的示例：

```
A = [3,6,7,4]
B = [2,8,9,3]
```

排序后：

```
A = [3,4,6,7]
B = [2,3,8,9]
```

排过序后，当A[i] < B[j]时，A[]中更接近B[j]的则可能出现在A[i]其后，因此i++；反过来，则需要j++

另：想起了牛顿法中寻找f(x) = 0，零点存在于f(x1) < 0, f(x2) > 0这个区间。在本题中，则是A[i1] < B[j], 那么如果有A[i2] > B[j]，则中间有可能就有这样的“零点”，也就是A[i]与B[j]差值的绝对值为零的情况。

Solution

```
public class Solution {
    /**
     * @param A, B: Two integer arrays.
     * @return: Their smallest difference.
     */
    public int smallestDifference(int[] A, int[] B) {
        if (A == null || B == null || A.length == 0 || B.length
== 0) {
            return 0;
        }

        Arrays.sort(A);
        Arrays.sort(B);

        int i = 0, j = 0;
        int minDiff = Integer.MAX_VALUE;
        while (i < A.length && j < B.length) {
            minDiff = Math.min(Math.abs(A[i] - B[j]), minDiff);
            if (A[i] < B[j]) {
                i++;
            } else {
                j++;
            }
        }
        return minDiff;
    }
}
```

Breadth-first Search

Depth-first Search

Reference

- Stanford CS97si
- BFS

Clone Graph

Question

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

How we serialize an undirected graph:

Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2. Second node is labeled as 1. Connect node 1 to node 2. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle. Visually, the graph looks like the following:

```
  1
 / \
/   \
0 --- 2
    / \
    \_/
```

Have you met this question in a real interview? Yes

Example return a deep copied graph.

Tags Breadth First Search Facebook

Analysis

本题是Graph相关的基础问题，图的遍历主要有两种方法：BFS，DFS，也就是广度优先搜索和深度优先搜索。

Breadth-first Search

> Wikipedia: “In graph theory, breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. Compare BFS with the equivalent, but more memory-efficient Iterative deepening depth-first search and contrast with depth-first search.”

这是一种对图的遍历方法，对于一个节点来说先把所有neighbors都检查一遍，再从第一个neighbor开始，循环往复。

由于BFS的这个特质，BFS可以帮助寻找最短路径。

通常BFS用queue+循环实现。

Depth-first Search

> Wikipedia: “Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.”

通常来说简便的DFS写法是用递归，如果非递归的话就是栈套迭代，思想是一样的。

思路1：使用BFS，先将头节点入queue，每一次queue出列一个node，然后检查这个node的所有的neighbors，如果没visited过，就入队，并更新neighbor。

思路2：使用DFS，可以分为迭代和循环两种方式，后者需要利用stack。

Solution

BFS - breadth first search, non-recursive

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new A
rrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
    /**
     * @param node: A undirected graph node
     * @return: A undirected graph node
     */
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
de) {
        if (node == null) {
            return null;
        }
        HashMap<UndirectedGraphNode, UndirectedGraphNode> hm = n
ew HashMap<UndirectedGraphNode, UndirectedGraphNode>();
        LinkedList<UndirectedGraphNode> queue = new LinkedList<U
ndirectedGraphNode>();
        UndirectedGraphNode head = new UndirectedGraphNode(node.
label);
        hm.put(node, head);
        queue.add(node);

        while (!queue.isEmpty()) {
            UndirectedGraphNode currentNode = queue.remove();
            for (UndirectedGraphNode neighbor : currentNode.neig
hbors) {
                if (!hm.containsKey(neighbor)) {
                    queue.add(neighbor);
                    UndirectedGraphNode newNeighbor = new Undire
ctedGraphNode(neighbor.label);
                    hm.put(neighbor, newNeighbor);
                }
```

```
                hm.get(currentNode).neighbors.add(hm.get(neighbo
r));
            }
        }

        return head;
    }
}
```

DFS - depth first search, non-recursive

```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new A
rrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
    /**
     * @param node: A undirected graph node
     * @return: A undirected graph node
     */
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode no
de) {
        if(node == null)
            return null;

        HashMap<UndirectedGraphNode, UndirectedGraphNode> hm = n
ew HashMap<UndirectedGraphNode, UndirectedGraphNode>();
        LinkedList<UndirectedGraphNode> stack = new LinkedList<U
ndirectedGraphNode>();
        UndirectedGraphNode head = new UndirectedGraphNode(node.
label);
        hm.put(node, head);
        stack.push(node);
```

```
        while(!stack.isEmpty()){
            UndirectedGraphNode curnode = stack.pop();
            for(UndirectedGraphNode aneighbor: curnode.neighbors
){//check each neighbor
                if(!hm.containsKey(aneighbor)){//if not visited,
then push to stack
                    stack.push(aneighbor);
                    UndirectedGraphNode newneighbor = new Undire
ctedGraphNode(aneighbor.label);
                    hm.put(aneighbor, newneighbor);
                }

                hm.get(curnode).neighbors.add(hm.get(aneighbor))
;
            }
        }

        return head;
    }
}
```

Reference

- Clone Graph leetcode java（DFS and BFS 基础）

N-Queens

Question

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

Have you met this question in a real interview? Yes

Example

There exist two distinct solutions to the 4-queens puzzle:

```
[
 // Solution 1
 [".Q..",
  "...Q",
  "Q...",
  "..Q."
 ],
 // Solution 2
 ["..Q.",
  "Q...",
  "...Q",
  ".Q.."
 ]
]
```

Challenge Can you do it without recursion?

Analysis

// TODO Add more 渐进式方法

Solution

```
import java.util.Arrays;
import java.util.ArrayList;

class Solution {
    /**
     * Get all distinct N-Queen solutions
     * @param n: The number of queens
     * @return: All distinct solutions
     * For example, A string '...Q' shows a queen on forth posit
ion
     */
    ArrayList<ArrayList<String>> solveNQueens(int n) {
        ArrayList<ArrayList<String>> result = new ArrayList<Arra
yList<String>>();
        String[] nQueens = new String[n];
        char[] init = new char[n];
        Arrays.fill(init, '.');
        Arrays.fill(nQueens, String.valueOf(Arrays.copyOf(init,
n)));
        search(n, 0, nQueens, result);
        return result;
    }


    private void search(int n, int row, String[] nQueens, ArrayL
ist<ArrayList<String>> result) {
        if (row == n) {
            result.add(new ArrayList<String>(Arrays.asList(nQuee
ns)));
            return;
        }

        for (int col = 0; col < n; col++) {
            if (isValid(nQueens, row, col, n)) {
                char[] chars;
```

```
                chars = nQueens[row].toCharArray();
                chars[col] = 'Q';
                nQueens[row] = String.valueOf(chars);
                // nQueens[row][col] = 'Q';
                search(n, row + 1, nQueens, result);
                chars = nQueens[row].toCharArray();
                chars[col] = '.';
                nQueens[row] = String.valueOf(chars);
                // nQueens[row][col] = '.';
            }
        }
    }

    private boolean isValid(String[] nQueens, int row, int col,
int n) {
        char[] chars;
        for (int i = 0; i < row; i++) {
            chars = nQueens[i].toCharArray();
            if (chars[col] == 'Q') {
                return false;
            }
        }

        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--
, j--) {
            chars = nQueens[i].toCharArray();
            if (chars[j] == 'Q') {
                return false;
            }
        }

        for (int  i = row - 1, j = col + 1; i >= 0 && j < n; i--
, j++) {
            chars = nQueens[i].toCharArray();
            if (chars[j] == 'Q') {
                return false;
            }
        }
        return true;
    }
```

```
    public static void main(String[] args) {
        Solution s = new Solution();
        ArrayList<ArrayList<String>> res = s.solveNQueens(Intege
r.valueOf(args[0]));
        // java Solution 4
        System.out.println(res);
        //[[.Q.., ...Q, Q..., ..Q.], [..Q., Q..., ...Q, .Q..]]
    }
};
```

Reference

- wikipedia: Eight Queens Puzzle
- LeetCode “全排列”问题系列(一) - 用交换元素法生成全排列及其应用，例题: Permutations I 和 II, N-Queens I 和 II，数独问题

Six Degrees

Question

Six degrees of separation is the theory that everyone and everything is six or fewer steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps.

Given a friendship relations, find the degrees of two people, return -1 if they can not been connected by friends of friends.

Example

Given a graph:

```
1------2-----4
 \           /
  \         /
   \--3--/
```

{1,2,3#2,1,4#3,1,4#4,2,3} and s = 1, t = 4 return 2

Given a graph:

```
1      2-----4
              /
            /
          3
```

{1#2,4#3,4#4,2,3} and s = 1, t = 4 return -1

Analysis

人际六度分离问题，本质是在Graph中寻找最短路径，那么利用Breadth-first Search(BFS)，通过计算s到t的路径长度，也就得到了degrees of two people。与最基础的BFS相比，利用hashmap存visited的节点的step，也就可以通过frontier节点得到邻近节点的steps(+1)。

Solution

```
/**
 * Definition for Undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     List<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) {
 *         label = x;
 *         neighbors = new ArrayList<UndirectedGraphNode>();
 *     }
 * };
 */
public class Solution {
    /**
     * @param graph a list of Undirected graph node
     * @param s, t two Undirected graph nodes
     * @return an integer
     */
    public int sixDegrees(List<UndirectedGraphNode> graph,
                          UndirectedGraphNode s,
                          UndirectedGraphNode t) {
        if (s == t) {
            return 0;
        }

        Map<UndirectedGraphNode, Integer> visited = new HashMap<
UndirectedGraphNode, Integer>();
        Queue<UndirectedGraphNode> queue = new LinkedList<Undire
ctedGraphNode>();

        queue.offer(s);
        visited.put(s, 0);
```

```
        while (!queue.isEmpty()) {
            UndirectedGraphNode frontier = queue.poll();
            int size = frontier.neighbors.size();
            for (int i = 0; i < size; i++) {
                UndirectedGraphNode node = frontier.neighbors.ge
t(i);
                if (visited.containsKey(node)) {
                    continue;
                }
                if (node == t) {
                    return visited.get(frontier) + 1;
                }
                queue.offer(node);
                visited.put(node, visited.get(frontier) + 1);
            }
        }
        return -1;
    }
}
```

Reference

- Breadth First Traversal for a Graph - GeeksforGeeks
- Emory University: Graph traversal algorithms: Breadth First Search

Number of Islands

`#DFS` `#UnionFind`

Question

Given a boolean 2D matrix, find the number of islands.

Notice

> 0 is represented as the sea, 1 is represented as the island. If two 1 is adjacent, we consider them in the same island. We only consider up/down/left/right adjacent.

Example Given graph:

```
[
  [1, 1, 0, 0, 0],
  [0, 1, 0, 0, 1],
  [0, 0, 0, 1, 1],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 1]
]
```

return 3.

Tags

Facebook Zenefits Google

Related Problems

Medium Surrounded Regions Hard Number of Islands II

Analysis

思路一： 此题可以考虑用Union Find，不过更简单的是用BFS或者DFS。 其中DFS结合mark的方法最巧妙简单，n^2循环，扫描 `grid[i][j]` , 如果是island的，即 `grid[i][j] == true` ，则计数加一（ans++），并对四个方向进行DFS查找，并将所有属于那坐岛屿的点mark为非岛屿。这样扫描过的地方全部会变成非岛屿，而岛屿的数量已被记录。

思路二： Union Find 利用Union Find的find和union操作，对岛屿进行合并计数。因为UnionFind结构一般来说是一维的

```
| 1 | 2 | 3 | 4 |
-----------------
| 2 | 2 | 2 | 4 |
```

表达了如下的连通结构

```
1 - 2    4
| /
3
```

因此可以转换二维矩阵坐标为一维数字，M *N* 的矩阵中`*(i,j)* -> *i* N + j`。

1. 建立UnionFind的parent[] (或者 parent hashmap)，初始化各个 `parent[i * N + j] = i * N + j` ，如果 `grid[i][j] == true` ，则岛屿计数count++
2. 之后再对矩阵遍历一次，当遇到岛屿时，则要检查上下左右四个方向的邻近点，如果也是岛屿则合并，并且count--；

Union Find结构可以用Path Compression和Weighted Union进行优化。

Solution

DFS (3ms AC)

```
public class Solution {
    /**
     * @param grid a boolean 2D matrix
     * @return an integer
     */
```

```
    public int numIslands(boolean[][] grid) {
        m = grid.length;
        if (m == 0) {
            return 0;
        }
        n = grid[0].length;
        if (n == 0) {
            return 0;
        }
        int nums = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == false) {
                    continue;
                }
                nums++;
                dfs(grid, i, j);
            }
        }
        return nums;
    }

    private void dfs(boolean[][] grid, int i, int j) {
        if (i < 0 || i >= m || j < 0 || j >= n) {
            return;
        }
        if (grid[i][j] == true) {
            grid[i][j] = false;
            dfs(grid, i - 1, j);
            dfs(grid, i + 1, j);
            dfs(grid, i, j - 1);
            dfs(grid, i, j + 1);
        }
    }

    private int m, n;
}
```

Union Find (Verbose, 19 ms AC)

```
class UnionFind {
    public int[] parent;
    public int[] size;

    // Initialize UnionFind
    public UnionFind() {}

    public UnionFind(int n) {
        this.parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            this.parent[i] = i;
            size[i] = 1;
        }
    }

    // Find Method
    public int find(int x) {
        return compressedFind(x);
    }

    // Find Method with Path Compression
    public int compressedFind(int x) {
        while (x != parent[x]) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    // Union Method with Weight
    public void union(int x, int y) {
        int parentX = find(x);
        int parentY = find(y);
        if (parentX == parentY) {
            return;
        }

        if (this.size[parentX] < this.size[parentY]) {
```

```
            this.size[parentY] += this.size[parentX];
            this.parent[parentX] = parentY;
        } else {
            this.size[parentX] += this.size[parentY];
            this.parent[parentY] = parentX;
        }
    }
}

class IslandUnionFind extends UnionFind {
    public int count;

    public void initIslands(boolean grid[][]) {
        count = 0;
        int m = grid.length;
        int n = grid[0].length;
        this.parent = new int[m * n];
        this.size = new int[m * n];

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == true) {
                    parent[i * n + j] = i * n + j;
                    count++;
                } else {
                    parent[i * n + j] = -1;
                }
                this.size[i * n + j] = 1;
            }
        }
    }

    public void updateCount(int k) {
        count = count + k;
    }

    public int getCount() {
        return count;
    }
}
```

```
public class Solution {

    public boolean insideGrid(int x, int y, int m, int n) {
        return (x >= 0 && y >= 0 && x < m && y < n);
    }
    /**
     * @param grid a boolean 2D matrix
     * @return an integer
     */
    public int numIslands(boolean[][] grid) {
        // Check Input
        if(grid==null || grid.length==0 || grid[0].length==0) {
            return 0;
        }

        IslandUnionFind uf = new IslandUnionFind();
        uf.initIslands(grid);

        int m = grid.length;
        int n = grid[0].length;

        // Helper Direction Array
        int[] dx = new int[] {-1, 1, 0, 0};
        int[] dy = new int[] {0, 0, -1, 1};

        // Row and Column Traversal
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (insideGrid(i, j, m, n) && grid[i][j]) {

                    // 4 directions for each point
                    for (int k = 0; k < 4; k++) {
                        int nx = i + dx[k];
                        int ny = j + dy[k];
                        if (insideGrid(nx, ny, m, n) && grid[nx]
[ny]) {
                            int cparent = uf.find(i * n + j);
                            int nparent = uf.find(nx * n + ny);
                            if (cparent != nparent) {
```

```
                                uf.union(cparent, nparent);
                                uf.updateCount(-1);
                            }
                        }
                    }
                }
            }

            return uf.getCount();
        }
    }
```

Solution

- programcreek: LeetCode – Number of Islands (Java)
- LeetCode Discussion: AC Java Solution using Union-Find with explanations

Word Search

Question

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example

Given board =

```
[
  "ABCE",
  "SFCS",
  "ADEE"
]
```

word = "ABCCED", -> returns true, word = "SEE", -> returns true, word = "ABCB", -> returns false.

Tags

Backtracking Facebook Depth First Search

Analysis

基本思路比较简单：从board上每一个点出发，用depth first search （DFS）上下左右四方向搜索匹配的word。需要注意到几点：

1. 考虑board的边界
2. cell是否已经被遍历过（临时转换为特定标识字符，比如'#'，recursion返回后再重置回原来的值，这样可以省去额外开辟二维数组visited[i][j]的空间复杂度)
3. 搜索结束的标志是 `k == word.length()`

Solution

DFS (15 ms 36.92% AC)

```
public class Solution {
    /**
     * @param board: A list of lists of character
     * @param word: A string
     * @return: A boolean
     */
    public boolean exist(char[][] board, String word) {
        if (board == null || board.length == 0 || board[0].lengt
h == 0) {
            return false;
        }
        if (word.length() == 0) {
            return true;
        }

        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] == word.charAt(0)) {
                    boolean result = dfs(board, word, i, j, 0);
                    if (result) {
                        return true;
                    }
                }
            }
        }
        return false;
    }

    private boolean dfs(char[][] board, String word, int i, int
j, int k) {
        if (board == null || board.length == 0 || board[0].lengt
h == 0) {
            return false;
        }
        if (word.length() == 0) {
```

```
            return true;
        }

        if (word.length() == k) {
            return true;
        }

        boolean result = false;
        if (insideBoard(board, i, j) && board[i][j] == word.char
At(k)) {
            char temp = board[i][j];
            board[i][j] = '#';
            result = dfs(board, word, i + 1, j, k + 1) ||
                dfs(board, word, i - 1, j, k + 1) ||
                dfs(board, word, i, j + 1, k + 1) ||
                dfs(board, word, i, j - 1, k + 1);
            board[i][j] = temp;
        }
        return result;
    }

    private boolean insideBoard(char[][] board, int i, int j) {
        return (i >= 0 && i < board.length && j >= 0 && j < boar
d[0].length);
    }
}
```

Reference

- LeetCode Discussion: Word Search
- 喜刷刷：Word Search
- programcreek: word search

Find the Connected Component in the Undirected Graph

Question

Find the number connected component in the undirected graph. Each node in the graph contains a label and a list of its neighbors. (a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.)

Notice

Each connected component should sort by label.

Clarification

Learn more about representation of graphs

Example

Given graph:

```
A------B  C
 \     |  |
  \    |  |
   \   |  |
    \  |  |
      D   E
```

Return {A,B,D}, {C,E}. Since there are two connected component which is {A,B,D}, {C,E}

Tags

`Breadth First Search` `Union Find`

Related Problems

- Medium Graph Valid Tree
- Medium Find the Weak Connected Component in the Directed Graph

Analysis

除了常规的BFS方法，还可以用Union Find。

1. 两重循环遍历所有的node，并存入一个HashSet（为什么是HashSet，HashMap是否可以？）
2. 用HashSet的元素初始化UnionFind（father HashMap的构建）
3. 再度两重循环遍历所有node，将now node 和 neighbor node全部union起来
4. 通过辅助函数print，对HashSet中的每一个节点进行父节点查找（find），把具有相同的父节点的子节点全部打包成ArrayList作为value，按照父节点的label为key，存入HashMap中，最后把这个HashMap的values打包成List，输出结果

Solution

```
/**
 * Definition for Undirected graph.
 * class UndirectedGraphNode {
 *     int label;
 *     ArrayList<UndirectedGraphNode> neighbors;
 *     UndirectedGraphNode(int x) { label = x; neighbors = new A
rrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
    // Define UnionFind Class
    class UnionFind{
        HashMap<Integer, Integer> father = new HashMap<Integer,
Integer>();
        UnionFind(HashSet<Integer> hashSet){
            for(Integer now : hashSet) {
                father.put(now, now);
            }
        }
```

```
        int find(int x){
            int parent =  father.get(x);
            while(parent!=father.get(parent)) {
                parent = father.get(parent);
            }
            return parent;
        }
        int compressed_find(int x){
            int parent =  father.get(x);
            while(parent!=father.get(parent)) {
                parent = father.get(parent);
            }
            int temp = -1;
            int fa = father.get(x);
            while(fa!=father.get(fa)) {
                temp = father.get(fa);
                father.put(fa, parent) ;
                fa = temp;
            }
            return parent;
        }
        void union(int x, int y){
            int fa_x = find(x);
            int fa_y = find(y);
            if(fa_x != fa_y)
            father.put(fa_x, fa_y);
        }
    }

    List<List<Integer>> print(HashSet<Integer> hashSet, UnionFin
d uf, int n) {
        List<List<Integer>> ans = new ArrayList<List<Integer>>()
;
        HashMap<Integer, List<Integer>> hashMap = new HashMap<In
teger, List<Integer>>();
        for (int i : hashSet) {
            int fa = uf.find(i);
            if (!hashMap.containsKey(fa)) {
                hashMap.put(fa, new ArrayList<Integer>());
            }
```

```
            List<Integer> now = hashMap.get(fa);
            now.add(i);
            hashMap.put(fa, now);
        }
        for (List<Integer> now : hashMap.values()) {
            Collections.sort(now);
            ans.add(now);
        }
        return ans;
    }

    /**
     * @param nodes a array of Undirected graph node
     * @return a connected set of a Undirected graph
     */
    public List<List<Integer>> connectedSet(ArrayList<Undirected
GraphNode> nodes) {
        HashSet<Integer> hashSet = new HashSet<Integer>();
        for (UndirectedGraphNode now : nodes) {
            hashSet.add(now.label);
            for (UndirectedGraphNode neighbor : now.neighbors) {
                hashSet.add(neighbor.label);
            }
        }

        UnionFind uf = new UnionFind(hashSet);

        for (UndirectedGraphNode now : nodes) {
            for (UndirectedGraphNode neighbor : now.neighbors) {
                int fnow = uf.find(now.label);
                int fneighbor = uf.find(neighbor.label);
                if (fnow != fneighbor) {
                    uf.union(now.label, neighbor.label);
                }
            }
        }
        return print(hashSet, uf, nodes.size());
    }
}
```

Reference

Find the Weak Connected Component in the Directed Graph

Question

Find the number Weak Connected Component in the directed graph. Each node in the graph contains a label and a list of its neighbors. (a connected set of a directed graph is a subgraph in which any two vertices are connected by direct edge path.)

Notice

Sort the element in the set in increasing order

Example

Given graph:

```
A----->B  C
 \     |  |
  \    |  |
   \   |  |
    \  v  v
     ->D  E <- F
```

Return {A,B,D}, {C,E,F}. Since there are two connected component which are {A,B,D} and {C,E,F}

Tags

Union Find

Related Problems

Hard Number of Islands II 16 % Medium Find the Weak Connected Component in the Directed Graph 22 % Medium Find the Connected Component in the Undirected Graph

Analysis

与Find the Connected Component in the Undirected Graph基本相同。

Solution

```
/**
* Definition for Directed graph.
* class DirectedGraphNode {
*     int label;
*     ArrayList<DirectedGraphNode> neighbors;
*     DirectedGraphNode(int x) { label = x; neighbors = new Arra
yList<DirectedGraphNode>(); }
* };
*/

public class Solution {
    class UnionFind{
        HashMap<Integer, Integer> father = new HashMap<Integer,
Integer>();
        UnionFind(HashSet<Integer> hashSet){
            for(Integer now : hashSet) {
                father.put(now, now);
            }
        }
        int find(int x){
            int parent =  father.get(x);
            while(parent!=father.get(parent)) {
                parent = father.get(parent);
            }
            return parent;
        }
        int compressed_find(int x){
            int parent =  father.get(x);
            while(parent!=father.get(parent)) {
                parent = father.get(parent);
            }
            int temp = -1;
```

```
            int fa = father.get(x);
            while(fa!=father.get(fa)) {
                temp = father.get(fa);
                father.put(fa, parent) ;
                fa = temp;
            }
            return parent;

        }

        void union(int x, int y){
            int fa_x = find(x);
            int fa_y = find(y);
            if(fa_x != fa_y)
            father.put(fa_x, fa_y);
        }
    }
    List<List<Integer> >  print(HashSet<Integer> hashSet, UnionF
ind uf, int n) {
        List<List <Integer> > ans = new ArrayList<List<Integer>>
();
        HashMap<Integer, List <Integer>> hashMap = new HashMap<I
nteger, List <Integer>>();
        for(int i : hashSet){
            int fa = uf.find(i);
            if(!hashMap.containsKey(fa)) {
                hashMap.put(fa,  new ArrayList<Integer>() );
            }
            List <Integer> now =  hashMap.get(fa);
            now.add(i);
            hashMap.put(fa, now);
        }
        for( List <Integer> now: hashMap.values()) {
            Collections.sort(now);
            ans.add(now);
        }
        return ans;
    }

    public List<List<Integer>> connectedSet2(ArrayList<DirectedG
```

```
raphNode> nodes){
        // Write your code here

        HashSet<Integer> hashSet = new HashSet<Integer>();
        for(DirectedGraphNode now : nodes){
            hashSet.add(now.label);
            for(DirectedGraphNode neighbour : now.neighbors) {
                hashSet.add(neighbour.label);
            }
        }
        UnionFind uf = new UnionFind(hashSet);


        for(DirectedGraphNode now : nodes){

            for(DirectedGraphNode neighbour : now.neighbors) {
                int fnow = uf.find(now.label);
                int fneighbour = uf.find(neighbour.label);
                if(fnow!=fneighbour) {
                    uf.union(now.label, neighbour.label);
                }
            }
        }


        return print(hashSet , uf, nodes.size());
    }

}
```

Graph Valid Tree

Question

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

Notice

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

Example

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Tags

`Depth First Search` Facebook Zenefits `Union Find` `Breadth First Search` Google

Related Problems

Medium Find the Connected Component in the Undirected Graph

Analysis

DFS, BFS均可以解此题，有趣的是Union Find的解法。// TODO: Add DFS, BFS

> This problem can be converted to finding a cycle in a graph. It can be solved by using DFS (Recursion) or BFS (Queue).

Union Find 思路

初始化Union Find的father map，让每一个节点的初始parent指向自己（自己跟自己是一个Group）；在循环读取edge list时，查找两个节点的parent，如果相同，说明形成了环（Cycle），那么这便不符合树（Tree）的定义，反之，如果不相同，则将其中一个节点设为另一个的parent，继续循环。

此外还有需要注意的是对于vertex和edge的validation，`|E| = |V| - 1`，也就是要验证`edges.length == n`，如果该条件不满足，则Graph一定不是valid tree。

Solution

Union Find Solution

```
public class Solution {
    class UnionFind {
        // Implemented with array instead of HashMap for simplic
ity
        int[] parent;

        // Constructor
        UnionFind (int n) {
            parent = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
            }
        }

        public int find(int x) {
            return compressed_find(x);
        }

        public int compressed_find(int x) {
            int x_parent = parent[x];
            while (x_parent != parent[x_parent]) {
                x_parent = parent[x_parent];
            }

            int temp = -1;
            int t_parent = parent[x];
```

```
            while (t_parent != parent[t_parent]) {
                temp = parent[t_parent];
                parent[t_parent] = x_parent;
                t_parent = temp;
            }
            return x_parent;
        }
        public void union(int x, int y) {
            int x_parent = find(x);
            int y_parent = find(y);
            if (x_parent != y_parent) {
                parent[x_parent] = y_parent;
            }
        }
    }
    /**
     * @param n an integer
     * @param edges a list of undirected edges
     * @return true if it's a valid tree, or false
     */
    public boolean validTree(int n, int[][] edges) {
        // Validation of |V| - 1 = |E|
        if (n - 1 != edges.length || n == 0) {
            return false;
        }
        UnionFind uf = new UnionFind(n);
        for (int i = 0; i < edges.length; i++) {
            if (uf.find(edges[i][0]) == uf.find(edges[i][1])) {
                return false;
            }
            uf.union(edges[i][0], edges[i][1]);
        }
        return true;
    }
}
```

DFS Solution

```
public boolean validTree(int n, int[][] edges) {
```

```
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integ
er, ArrayList<Integer>>();
    for(int i=0; i<n; i++){
        ArrayList<Integer> list = new ArrayList<Integer>();
        map.put(i, list);
    }

    for(int[] edge: edges){
        map.get(edge[0]).add(edge[1]);
        map.get(edge[1]).add(edge[0]);
    }

    boolean[] visited = new boolean[n];

    if(!helper(0, -1, map, visited))
        return false;

    for(boolean b: visited){
        if(!b)
            return false;
    }

    return true;
}

public boolean helper(int curr, int parent, HashMap<Integer, Arr
ayList<Integer>> map, boolean[] visited){
    if(visited[curr])
        return false;

    visited[curr] = true;

    for(int i: map.get(curr)){
        if(i!=parent && !helper(i, curr, map, visited)){
            return false;
        }
    }

    return true;
}
```

BFS Solution

```
public boolean validTree(int n, int[][] edges) {
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integ
er, ArrayList<Integer>>();
    for(int i=0; i<n; i++){
        ArrayList<Integer> list = new ArrayList<Integer>();
        map.put(i, list);
    }

    for(int[] edge: edges){
        map.get(edge[0]).add(edge[1]);
        map.get(edge[1]).add(edge[0]);
    }

    boolean[] visited = new boolean[n];

    LinkedList<Integer> queue = new LinkedList<Integer>();
    queue.offer(0);
    while(!queue.isEmpty()){
        int top = queue.poll();
        if(visited[top])
            return false;

        visited[top]=true;

        for(int i: map.get(top)){
            if(!visited[i])
                queue.offer(i);
        }
    }

    for(boolean b: visited){
        if(!b)
            return false;
    }

    return true;
}
```

Reference

- LeetCode Discuss: Java Union-Find solution
- Program Creek: Graph Valid Tree (Java) DFS & BFS
- Princeton CS: Union Find

Surrounded Regions

Question

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O''s into 'X''s in that surrounded region.

Example

```
X X X X
X O O X
X X O X
X O X X
```

After capture all regions surrounded by 'X', the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Tags

Breadth First Search Union Find

Related Problems

Hard Number of Islands II Easy Number of Islands

Analysis

本题也是一个多解的问题。

一种比较巧妙的思路是从边缘的'O'出发，通过DFS，所有能够遍历的'O'都可以暂时被标记为'#'，那么剩下未能被标记的'O'说明被surrounded，需要在遍历结束之后全部转为'X'。（用DFS或者BFS要警惕栈溢出）

另一种方法是用Union Find，将与边缘相连通的'O'全部union到一个dummy node（也可以用hasEdge[]来存储，不过内存占用更多），最终将没有和这个dummy node是一个component的'O'点全部标记为'X'。

Solution

Union Find (20 ms 14.75% AC)

```
class UnionFind {
    private int[] id;
    private int[] size;

    public UnionFind(int n) {
        id = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            id[i] = i;
            size[i] = 1;
        }
    }

    public int find(int i) {
        while (i != id[i]) {
            id[i] = id[id[i]];
            i = id[i];
        }
        return i;
    }

    public void union(int x, int y) {
        int i = find(x);
        int j = find(y);
        if (i == j) {
            return;
        }
```

```
            if (size[i] < size[j]) {
                id[i] = j;
                size[j] += size[i];
            } else {
                id[j] = i;
                size[i] += size[j];
            }
        }

        public boolean connected(int x, int y) {
            return find(x) == find(y);
        }
    }

    public class Solution {

        private boolean isEdge(int M, int N, int i, int j) {
            return (i == 0 || i == M - 1 || j == 0 || j == N - 1);
        }

        private boolean insideBoard(int M, int N, int i, int j) {
            return (i < M && i >= 0 && j < N && j >= 0);
        }
        /**
         * @param board a 2D board containing 'X' and 'O'
         * @return void
         */
        public void surroundedRegions(char[][] board) {
            if (board == null || board.length == 0 || board[0].lengt
    h == 0) {
                return;
            }
            int M = board.length;
            int N = board[0].length;

            int[] dirX = {0, 0, -1, 1};
            int[] dirY = {-1, 1, 0, 0};

            UnionFind uf = new UnionFind(M * N + 1);
```

```
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                if (board[i][j] == 'O') {
                    if (isEdge(M, N, i, j)) {
                        uf.union(i * N + j, M * N);
                    } else {
                        for (int k = 0; k < 4; k++) {
                            int x = i + dirX[k];
                            int y = j + dirY[k];
                            if (insideBoard(M, N, x, y) && board
[x][y] == 'O') {
                                uf.union(i * N + j, x * N + y);
                            }
                        }
                    }
                }
            }
        }

        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                if (!uf.connected(i * N + j, M * N)) {
                    board[i][j] = 'X';
                }
            }
        }

    }
}
```

Reference

- LeetCode Discussion: Solve it using Union Find
- Flood Fill Algorithm
- 水中的鱼 Surrounded Regions, Solution BFS
- 喜刷刷 Surrounded Regions
- programcreek: LeetCode – Surrounded Regions

Segment Tree

Segment Tree (a.k.a Interval Tree) is an advanced data structure which can support queries like:

- which of these intervals contain a given point
- which of these points are in a given interval

See wiki:

- Segment Tree
- Interval Tree

The structure of Segment Tree is a binary tree which each node has two attributes start and end denote an segment / interval.

start and end are both integers, they should be assigned in following rules:

- The root's start and end is given by build method.
- The left child of node A has start=A.left, end=(A.left + A.right) / 2.
- The right child of node A has start=(A.left + A.right) / 2 + 1, end=A.right. if start equals to end, there will be no children for this node.
- Implement a build method with two parameters start and end, so that we can create a corresponding segment tree with every node has the correct start and end value, return the root of this segment tree.

Example Given start=0, end=3. The segment tree will be:

```
                [0,  3]
              /        \
       [0,  1]           [2, 3]
       /     \           /     \
  [0, 0]  [1, 1]     [2, 2]  [3, 3]
```

Given start=1, end=6. The segment tree will be:

```
                [1,  6]
              /         \
       [1,  3]           [4,  6]
       /     \           /     \
    [1, 2]  [3,3]     [4, 5]   [6,6]
    /    \           /     \
 [1,1]   [2,2]     [4,4]   [5,5]
```

Solution

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end) {
 *         this.start = start, this.end = end;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     *@param start, end: Denote an segment / interval
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int start, int end) {
        // write your code here
        if(start > end) {  // check core case
            return null;
        }

        SegmentTreeNode root = new SegmentTreeNode(start, end);

        if(start != end) {
            int mid = (start + end) / 2;
            root.left = build(start, mid);
            root.right = build(mid + 1, end);

            // root.max = Math.max(root.left.max, root.right.max
);
        }
        return root;
    }
}
```

Segment Tree Build

Question

The structure of Segment Tree is a binary tree which each node has two attributes start and end denote an segment / interval.

start and end are both integers, they should be assigned in following rules:

- The root's start and end is given by build method.
- The left child of node A has start=A.left, end=(A.left + A.right) / 2.
- The right child of node A has start=(A.left + A.right) / 2 + 1, end=A.right. if start equals to end, there will be no children for this node.

Implement a build method with two parameters start and end, so that we can create a corresponding segment tree with every node has the correct start and end value, return the root of this segment tree.

Example Given start=0, end=3. The segment tree will be:

```
          [0,  3]
        /        \
[0,  1]           [2, 3]
/     \           /     \
```

[0, 0] [1, 1] [2, 2] [3, 3] Given start=1, end=6. The segment tree will be:

```
          [1,  6]
        /        \
[1,  3]           [4,  6]
/     \           /     \
```

[1, 2] [3,3] [4, 5] [6,6] / \ / \ [1,1] [2,2] [4,4] [5,5]

Analysis

求出中间值，再分别对left，right进行build，递归调用build，生成子树。

Solution

```
/**
 * Definition of SegmentTreeNode:
 * public class SegmentTreeNode {
 *     public int start, end;
 *     public SegmentTreeNode left, right;
 *     public SegmentTreeNode(int start, int end) {
 *         this.start = start, this.end = end;
 *         this.left = this.right = null;
 *     }
 * }
 */
public class Solution {
    /**
     *@param start, end: Denote an segment / interval
     *@return: The root of Segment Tree
     */
    public SegmentTreeNode build(int start, int end) {
        if(start > end) {  // check core case
            return null;
        }

        SegmentTreeNode root = new SegmentTreeNode(start, end);

        if(start != end) {
            int mid = (start + end) / 2;
            root.left = build(start, mid);
            root.right = build(mid + 1, end);
        }
        return root;
    }
}
```

Trie

Reference

- The Trie: A Neglected Data Structure
- ProgramCreek: LeetCode – Implement Trie (Prefix Tree) (Java)

Implement Trie (Prefix Tree)

Question

Implement a trie with insert, search, and startsWith methods.

Note:

You may assume that all inputs are consist of lowercase letters `a-z`.

Analysis

Solution

Implementation 1

A trie node should contains the character, its children and the flag that marks if it is a leaf node. You can use this diagram to walk though the Java solution.

```
class TrieNode {
    // Initialize your data structure here.
    char c;
    HashMap<Character, TrieNode> children = new HashMap<Characte
r, TrieNode>();
    boolean isLeaf;

    public TrieNode() {}

    public TrieNode(char c) {
        this.c = c;
    }
}

public class Trie {
    private TrieNode root;
```

```
    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        HashMap<Character, TrieNode> children = root.children;

        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);

            TrieNode t;
            if (children.containsKey(c)) {
                t = children.get(c);
            } else {
                t = new TrieNode(c);
                children.put(c, t);
            }

            children = t.children;

            // Set leaf node
            if (i == word.length() - 1) {
                t.isLeaf = true;
            }
        }
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
        TrieNode t = searchNode(word);

        if (t != null && t.isLeaf) {
            return true;
        } else {
            return false;
        }
    }
```

```
    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        if (searchNode(prefix) == null) {
            return false;
        } else {
            return true;
        }
    }

    // Search a String in Trie, return TrieNode if exists
    public TrieNode searchNode(String str) {
        HashMap<Character, TrieNode> children = root.children;
        TrieNode t = null;
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            if (children.containsKey(c)) {
                t = children.get(c);
                children = t.children;
            } else {
                return null;
            }
        }
        return t;
    }
}

// Your Trie object will be instantiated and called as such:
// Trie trie = new Trie();
// trie.insert("somestring");
// trie.search("key");
```

Java Implementation 2

Each trie node can only contains 'a'-'z' characters. So we can use a small array to store the character.

```
class TrieNode {
    TrieNode[] arr;
    boolean isEnd;
```

```
    // Initialize your data structure here.
    public TrieNode() {
        this.arr = new TrieNode[26];
    }

}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        TrieNode p = root;
        for(int i=0; i<word.length(); i++){
            char c = word.charAt(i);
            int index = c-'a';
            if(p.arr[index]==null){
                TrieNode temp = new TrieNode();
                p.arr[index]=temp;
                p = temp;
            }else{
                p=p.arr[index];
            }
        }
        p.isEnd=true;
    }

    // Returns if the word is in the trie.
    public boolean search(String word) {
        TrieNode p = searchNode(word);
        if(p==null){
            return false;
        }else{
            if(p.isEnd)
                return true;
        }
```

```
        return false;
    }

    // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        TrieNode p = searchNode(prefix);
        if(p==null){
            return false;
        }else{
            return true;
        }
    }

    public TrieNode searchNode(String s){
        TrieNode p = root;
        for(int i=0; i<s.length(); i++){
            char c= s.charAt(i);
            int index = c-'a';
            if(p.arr[index]!=null){
                p = p.arr[index];
            }else{
                return null;
            }
        }

        if(p==root)
            return null;

        return p;
    }
}
```

Reference

- programcreek: LeetCode – Implement Trie (Prefix Tree) (Java)

Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

```
void addWord(word)
boolean search(word)
```

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Note: You may assume that all words are consist of lowercase letters a-z.

Analysis

在Implement Trie的基础之上，应用Trie。不同之处在于对于pattern的匹配，需要进行backtracking，也就是利用DFS，对于匹配了'.'的那个TrieNode的每一个children都循环遍历，如果有一个为true，则说明找到一个match，即可返回true。

Solution

```
class TrieNode {
    public boolean isLeaf;
    public TrieNode[] children;
```

```
    public TrieNode() {
        this.children = new TrieNode[26];
    }
}

public class WordDictionary {

    private TrieNode root;

    public WordDictionary() {
        this.root = new TrieNode();
    }

    // Adds a word into the data structure.
    public void addWord(String word) {
        TrieNode p = this.root;
        for (int i = 0; i < word.length(); i++) {
            int index = word.charAt(i) - 'a';
            if (p.children[index] == null) {
                TrieNode temp = new TrieNode();
                p.children[index] = temp;
            }
            p = p.children[index];
        }
        p.isLeaf = true;
    }

    // Returns if the word is in the data structure. A word could

    // contain the dot character '.' to represent any one letter.

    public boolean search(String word) {
        return match(this.root, word, 0);
    }

    private boolean match(TrieNode p, String word, int k) {
        if (k == word.length()) {
            return p.isLeaf;
        }
```

```
        if (word.charAt(k) == '.') {
            for (int i = 0; i < p.children.length; i++) {
                if (p.children[i] != null) {
                    if (match(p.children[i], word, k + 1)) {
                        return true;
                    }
                }
            }
        } else {
            int index = word.charAt(k) - 'a';
            return p.children[index] != null && match(p.children
[index], word, k + 1);
        }
        return false;
    }
}

// Your WordDictionary object will be instantiated and called as
 such:
// WordDictionary wordDictionary = new WordDictionary();
// wordDictionary.addWord("word");
// wordDictionary.search("pattern");
```

Reference

- LeetCode Discuss
- ProgramCreek Solution
- Jiuzhang Solution

Word Search II

Question

Given a matrix of lower alphabets and a dictionary. Find all words in the dictionary that can be found in the matrix. A word can start from any position in the matrix and go left/right/up/down to the adjacent position.

Example

Given matrix:

```
doaf
agai
dcan
```

and dictionary:

`{"dog", "dad", "dgdg", "can", "again"}``

return `{"dog", "dad", "can", "again"}`

dog:

```
doaf
agai
dcan
```

dad:

```
doaf
agai
dcan
```

can:

```
doaf
agai
dcan
```

again:

```
doaf
agai
dcan
```

Challenge

Using trie to implement your algorithm.

Tags

LintCode Copyright Airbnb Trie Backtracking

Another Version

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example, Given words = `["oath","pea","eat","rain"]` and board =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return `["eat","oath"]`.

Note:

You may assume that all inputs are consist of lowercase letters a-z.

Hint:

> You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?
>
> If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

Analysis

这一题的基本思路与Word Search是相似的，都是需要寻找在一个alphabet soup中是否存在word，可以用DFS recursion的办法来寻找。不过对于Word Search II，因为事先存在一个words dictionary，所以可以利用Trie的数据结构来存储，提高查询效率。

Solution

```
class TrieNode {
    char c;
    HashMap<Character, TrieNode> children;
    boolean isLeaf;
    String s;

    public TrieNode() {
        this.children = new HashMap<Character, TrieNode>();
        this.s = "";
    }

    public TrieNode(char c) {
```

```
        this.c = c;
        this.s = "";
        this.children = new HashMap<Character, TrieNode>();
        this.isLeaf = false;
    }
}

class Trie {
    public TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        HashMap<Character, TrieNode> children = root.children;
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);

            TrieNode t;
            if (children.containsKey(c)) {
                t = children.get(c);
            } else {
                t = new TrieNode(c);
                children.put(c, t);
            }

            children = t.children;

            if (i == word.length() - 1) {
                t.isLeaf = true;
                t.s = word;
            }
        }
    }

    public boolean search(String word) {
        TrieNode t = searchTrieNode(word);
        if (t != null && t.isLeaf) {
            return true;
```

```
        } else {
            return false;
        }
    }

    public TrieNode searchTrieNode(String str) {
        HashMap<Character, TrieNode> children = root.children;
        TrieNode t = null;

        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            if (children.containsKey(c)) {
                t = children.get(c);
                children = t.children;
            } else {
                return null;
            }
        }

        return t;
    }
}

public class Solution {
    int[] dirX = {0, 0, 1, -1};
    int[] dirY = {-1, 1, 0, 0};

    public void searchWord(char[][] board, int i, int j, TrieNod
e root, ArrayList<String> ans) {
        if (root.isLeaf == true) {
            if (!ans.contains(root.s)) {
                ans.add(root.s);
            }
        }
        if (i >= 0 && i < board.length &&
            j >= 0 && j < board[0].length &&
            board[i][j] != '#' && root != null) {

            if (root.children.containsKey(board[i][j])) {
                for (int k = 0; k < 4; k++) {
```

```
                    int x = i + dirX[k];
                    int y = j + dirY[k];
                    char temp = board[i][j];
                    board[i][j] = '#';
                    searchWord(board, x, y, root.children.get(te
mp), ans);
                    board[i][j] = temp;
                }
            }
        }
    }
    /**
     * @param board: A list of lists of character
     * @param words: A list of string
     * @return: A list of string
     */
    public ArrayList<String> wordSearchII(char[][] board, ArrayL
ist<String> words) {
        ArrayList<String> ans = new ArrayList<String>();

        Trie trie = new Trie();
        for (String word : words) {
            trie.insert(word);
        }

        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                searchWord(board, i, j, trie.root, ans);
            }
        }

        return ans;
    }
}
```

Reference

- Jiuzhang Word Search II

不积跬步无以至千里，不积小流无以成江海。

练习，思考，总结。

1. 总结归类相似题目
2. 找出适合同一类题目的模板程序
3. 熟练掌握基础题
4. 从题海中找到知识网络，看到本质

参考资料

算法总结

- 编程之法：面试和算法心得 (The-Art-Of-Programming-By-July)
- July CSDN Blog
- Problem Solving with Algorithms and Data Structures Using Python
- Top 10 Algorithms for Coding Interview - Program Creek pdf
- Top 10 algorithms in Interview Questions - GeeksforGeeks
- Code_Ganker LeetCode总结 | LeetCode总结
- Logan's LintCode总结
- Programming Interview Questions - Arden Dertat

博客文章

- 我的算法学习之路 by Lucida@Google
- 程序员必读书单 1.0 by Lucida@Google
- 教你如何迅速秒杀掉：99%的海量数据处理面试题
- 十道海量数据处理面试题与十个方法大总结
- 程序员面试、算法研究、编程艺术、红黑树、数据挖掘5大系列集锦

可视化

- VisuAlgo - visualising data structures and algorithms through animation
- Data Structure Visualizations

刷题记录

- Zhang Lei: 水中的鱼
- yuanbin - 数据结构与算法/leetcode/lintcode题解
- tanglei
- Yu's Coding Garden
- siddontang - LeetCode题解
- Cracking the Coding Interview 6th Ed. Solutions
- Code Ganker征服代码
- lefttree LintCode 题解
- ShawnFan LintCode
- *小土刀的面试刷题笔记

Online Judge

- LeetCode
- LintCode
- InterviewBit
- HackerRank
- GeeksforGeeks Practice

在线课程

- CS 97SI: Introduction to Programming Contests

其他

- Runestone Interactive
- How To Think Like a Computer Scientist
- CS Principles: Big Ideas in Programming
- Problem Solving with Algorithms and Data Structures